Inexact String Matching

Professor Sean Goggins

Examples from "geeksforgeeks.org", and seangoggins.net

Algorithms

- Edit Distance
- Hirschberg's algorithm for Linear Space
- General and affine gap penalties for edit distance
- Four Russians speedup for edit distance

Edit Distance

- If you have two strings (str1 and str2) you need to find the minimum number of edit operations for converting str1 to str2.
  - insert
  - remove
  - replace
- Each operation has an equal cost

Example Problems

Input: str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.

Input: str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.

Input: str1 = "sunday", str2 = "saturday"
Output: 3
Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.
Replace 'n' with 'r', insert t, insert a

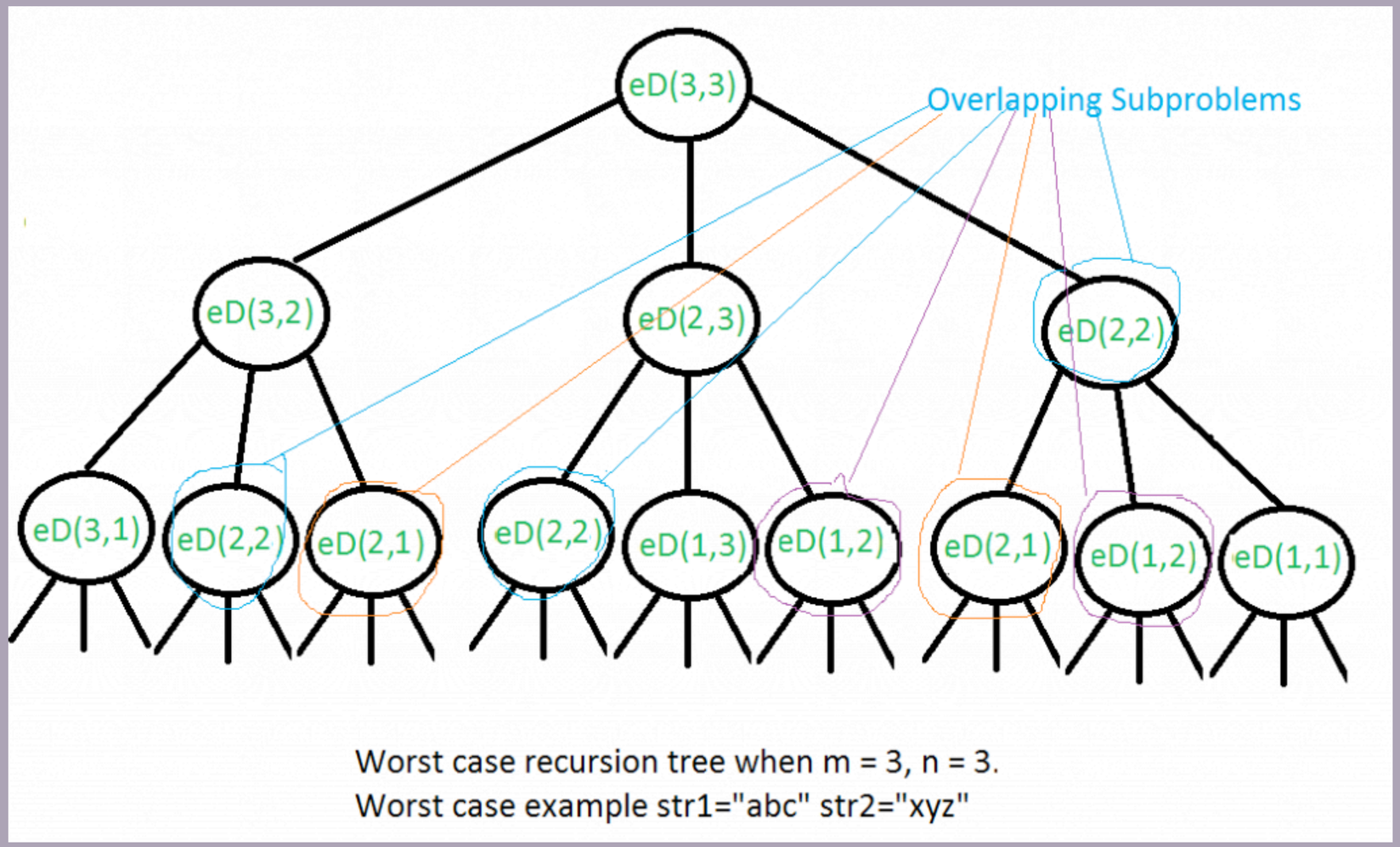
Psuedo-ish Code

- Process all characters one by one. You can start from the left or right side of both strings.
- From the right corner, there are two possibilities for each string traversal:
  - m: Length of str1 (first string)
  - n: Length of str2 (second string)
- If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
- Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
  - Insert: Recur for m and n-1
  - Remove: Recur for m-1 and n
  - Replace: Recur for m-1 and n-1

Psuedo-ish Code

```
def editDistance(str1, str2, m, n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m == 0:
        return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n == 0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1] == str2[n-1]:
        return editDistance(str1, str2, m-1, n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1),    # Insert
                   editDistance(str1, str2, m-1, n),    # Remove
                   editDistance(str1, str2, m-1, n-1)    # Replace
                   )


# Driver code
str1 = "sunday"
str2 = "saturday"
print (editDistance(str1, str2, len(str1), len(str2)))

# This code is contributed by Bhavya Jain
```

Time Complexity



Worst case recursion tree when m = 3, n = 3.
Worst case example str1="abc" str2="xyz"

Python Code

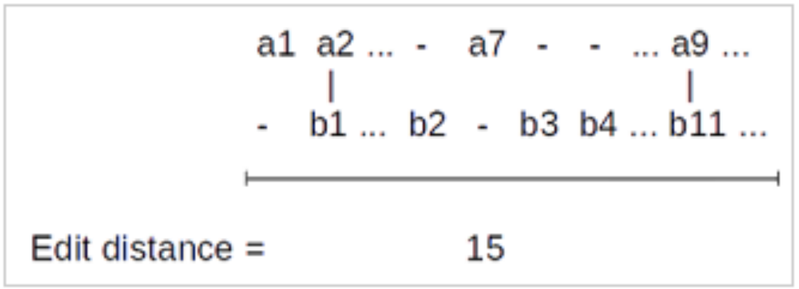
```
def editDistDP(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]

    # Fill d[][] in bottom up manner
    for i in range(m + 1):
        for j in range(n + 1):

            # If first string is empty, only option is to
            # insert all characters of second string
            if i == 0:
                dp[i][j] = j    # Min. operations = j

            # If second string is empty, only option is to
            # remove all characters of second string
            elif j == 0:
                dp[i][j] = i    # Min. operations = i

            # If last characters are same, ignore last char
            # and recur for remaining string
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]

            # If last character are different, consider all
            # possibilities and find minimum
            else:
                dp[i][j] = 1 + min(dp[i][j-1],        # Insert
                                   dp[i-1][j],        # Remove
                                   dp[i-1][j-1])    # Replace

    return dp[m][n]


# Driver code
str1 = "sunday"
str2 = "saturday"

print(editDistDP(str1, str2, len(str1), len(str2)))
```

Hirschberg's algorithm for Linear Space

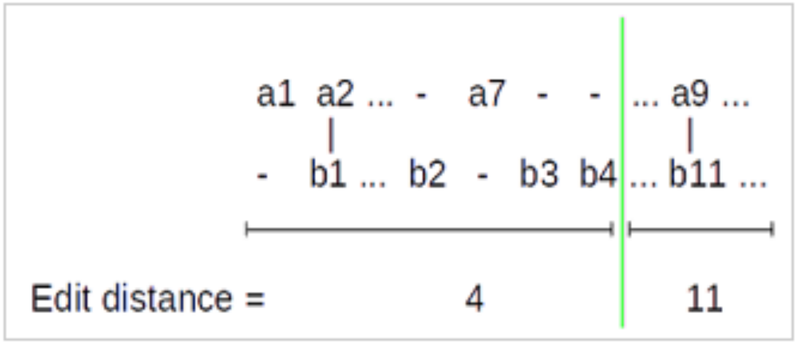
Hirschberg's algorithm lets us compute the whole alignment using time O(n*m) and space O(m+n). It uses NW' algorithm and divide and conquer approach:

1. if problem is trivial, compute it:
   - if n or m is 0 then there is n or m insertions or deletions
   - if n = 1 then there is m-1 insertions or deletions and 1 match or change
2. if problem is bigger, divide it into 2 smaller, independent problems
   - divide L in the middle. into 2 sublists L1, L2
   - find optimum division R = R1 R2
   - recursively find the alignment of L1 and R1
   - recursively find the alignment of L2 and R2
   - concatenate results

Of course point 2. needs better explanation. How can we divide big problem into smaller ones? Let's imagine some optimal alignment between L = a1...a27 and R = b1...b32:

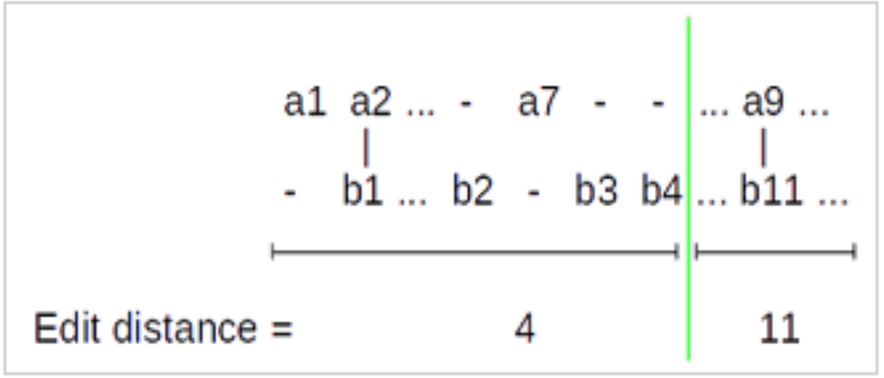


It means we can transform L into R in 15 operations (to be more precise: cost of transformation is 15). Obviously we can divide this alignment at any arbitrary position into 2 independent alignments, for example:



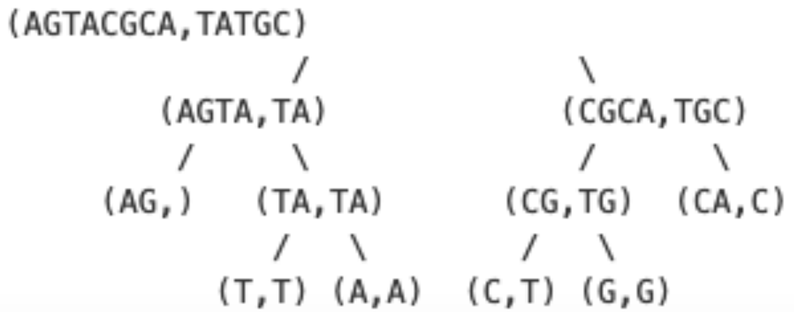
Above edit distances is of course just an example. The point is, they must add up to the overall edit distance (15).

It means we can transform L into R in 15 operations (to be more precise: cost of transformation is 15). Obviously we can divide this alignment at any arbitrary position into 2 independent alignments, for example:

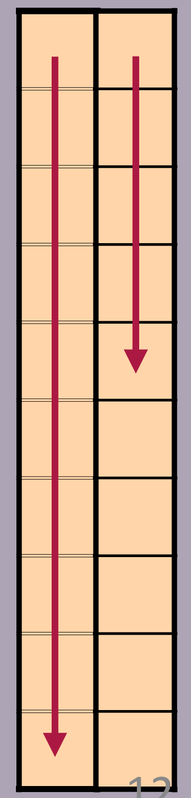


Above edit distances is of course just an example. The point is, they must add up to the overall edit distance (15).

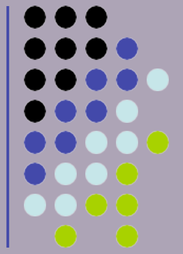
So now we have L = (a1...a7) (a8...a27) = L1 L2 and R = (b1...b4) (a8...a32) = R1 R2. It means we can transform L1 into R1 in 4 operations and L2 into R2 in 11 operations. We can concatenate those transformations to get the overall result. It means that if we somehow know the right division we'll be able to compute the alignment of L with R by computing alignments of L1 with R1 and L2 with R2 and simply concatenating results. So it will be possible to split big problem recursively until we reach trivial cases. For example (from wikipedia) to align AGTACGCA and TATGC we could do:



Linear-Space Alignment



Subsequences and Substrings



Definition A string x' is a substring of a string x,

if x = ux'v for some prefix string u and suffix string v

(similarly, x' = $x_i \ldots x_j$, for some $1 \le i \le j \le |x|$)

A string x' is a subsequence of a string x

if x' can be obtained from x by deleting 0 or more letters

(x' = $x_{i1} \ldots x_{ik}$, for some $1 \le i_1 \le \ldots \le i_k \le |x|$) Note: a

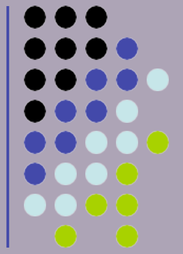
substring is always a subsequence

Example: **x = abracadabra**

y = cadabr; ***substring***

z = brcdbr; ***subseqence, not substring***

Hirschberg's algortihm

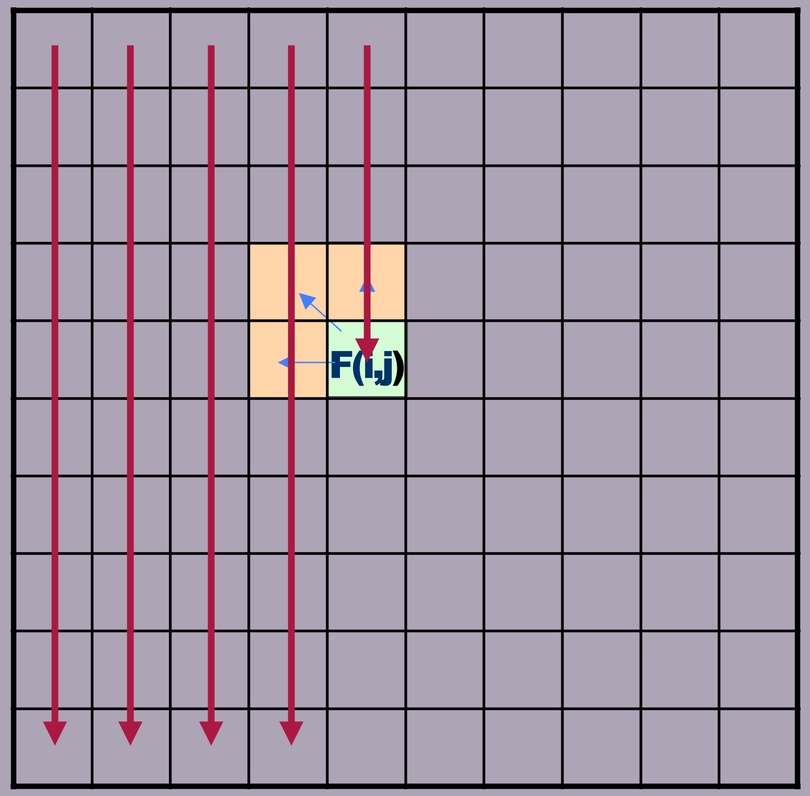


Given a set of strings x, y,…, a common subsequence is a string u that is a subsequence of all strings x, y, …

- Longest common subsequence
  - Given strings $x = x_1 x_2 \dots x_M$, $y = y_1 y_2 \dots y_N$,
  - Find longest common subsequence $u = u_1 \dots u_k$
- Algorithm:
  - $F(i, j) = \max \begin{cases} F(i-1, j) \\ F(i, j-1) \\ F(i-1, j-1) + [1, \text{if } x_i = y_j; 0 \text{ otherwise}] \end{cases}$
  - Ptr(i, j) = (same as in N-W)
  - Termination: trace back from Ptr(M, N), and prepend a letter to u whenever
    - Ptr(i, j) = DIAG and $F(i-1, j-1) < F(i, j)$
- Hirschberg's original algorithm solves this in linear space

Introduction: Compute optimal score

It is easy to compute F(M, N) in linear space



Allocate (column[1]) Allocate (column[2])

For i = 1....M

If i > 1, then:

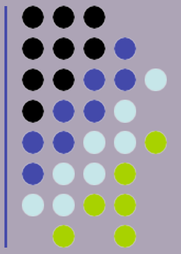
Free(column[i – 2])

Allocate(column[i])

For j = 1...N

F(i, j) = ...

Linear-space alignment



To compute both the optimal score and the optimal alignment:

Divide & Conquer approach:

Notation:

x^r, y^r: reverse of x, y

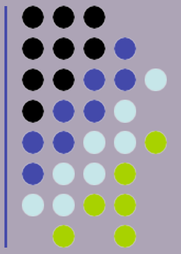
E.g. x = accgg;

x^r = ggcca

$F^r(i, j)$: optimal score of aligning $x^r_1 \ldots x^r_i$ & $y^r_1 \ldots y^r_j$

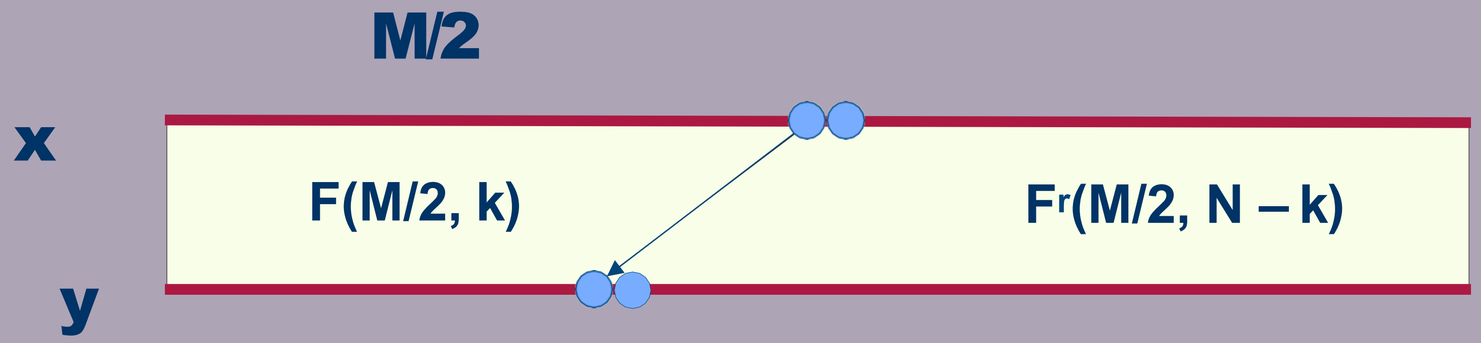
same as aligning $x_{M-i+1} \ldots x_M$ & $y_{N-j+1} \ldots y_N$

Linear-space alignment



Lemma: (assume M is even)

$$F(M, N) = \max_{k=0\ldots N}(F(M/2, k) + F^r(M/2, N - k))$$

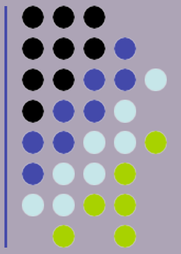


Example:

```
ACC-GGTGCCCAGGACTG--CAT
ACCAGGTG----GGACTGGGCAG
```

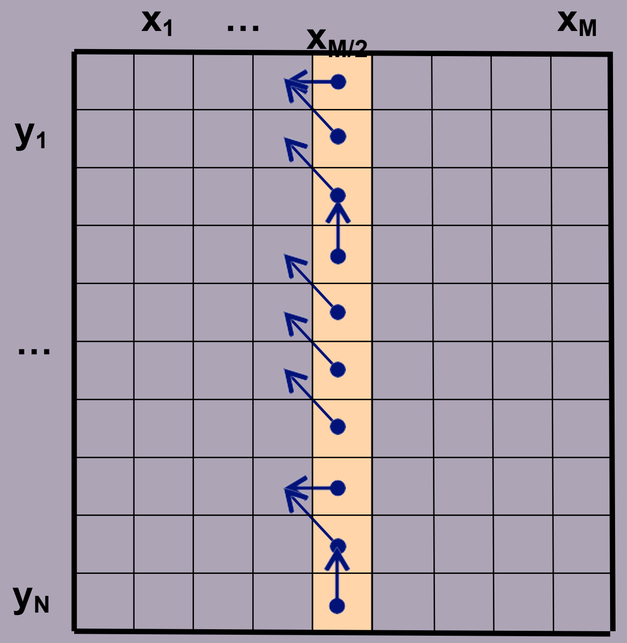
$k^* = 8$

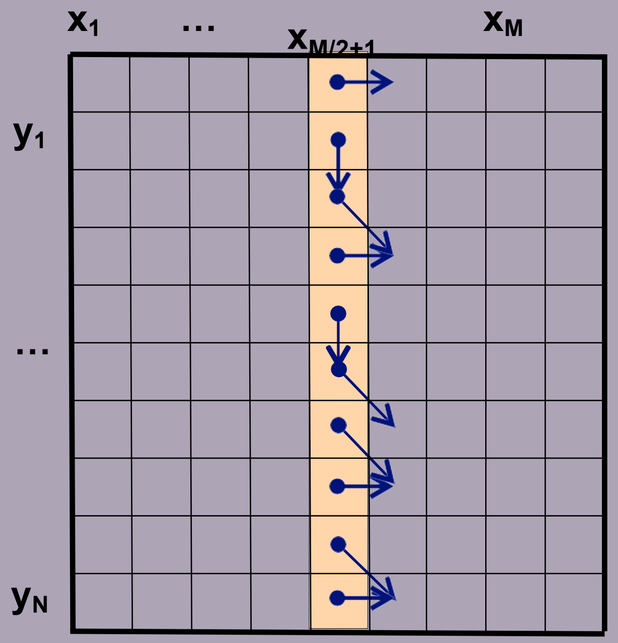
Linear-space alignment



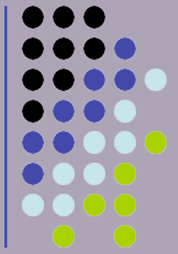
- Now, using 2 columns of space, we can compute for k = 1…M, F(M/2, k), F^r(M/2, N – k)

PLUS the backpointers

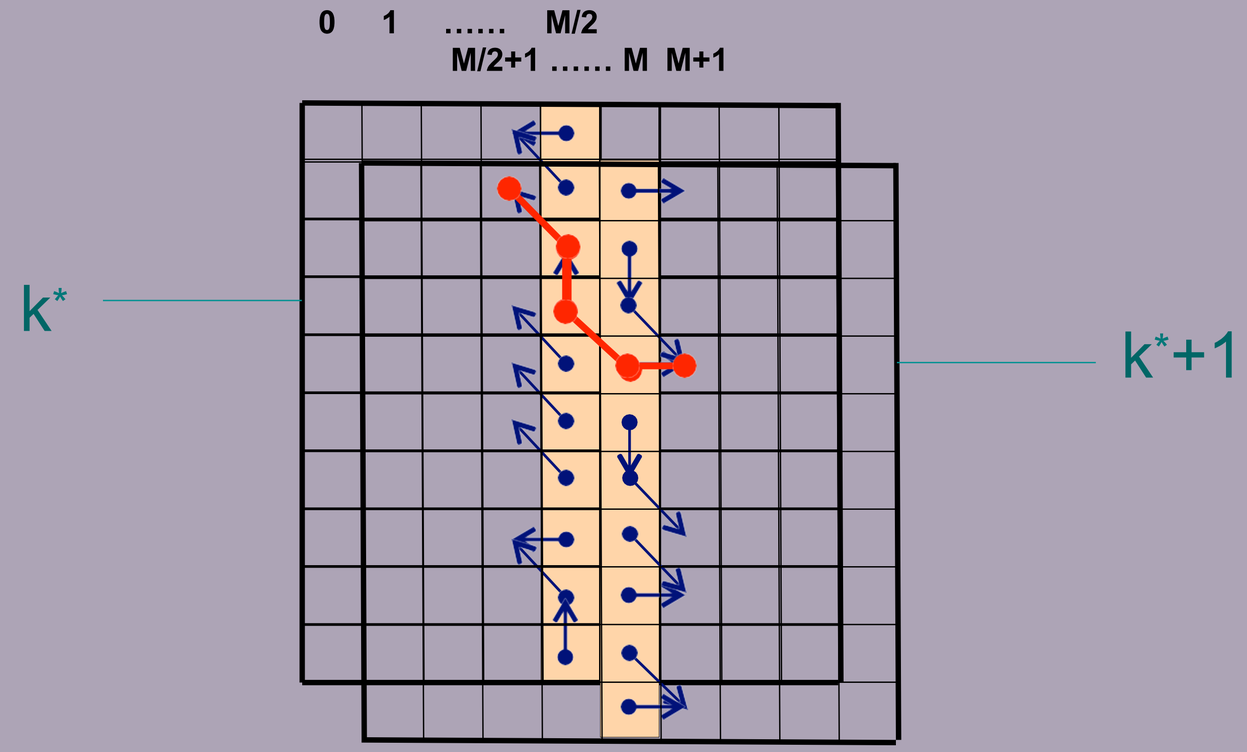




Linear-space alignment

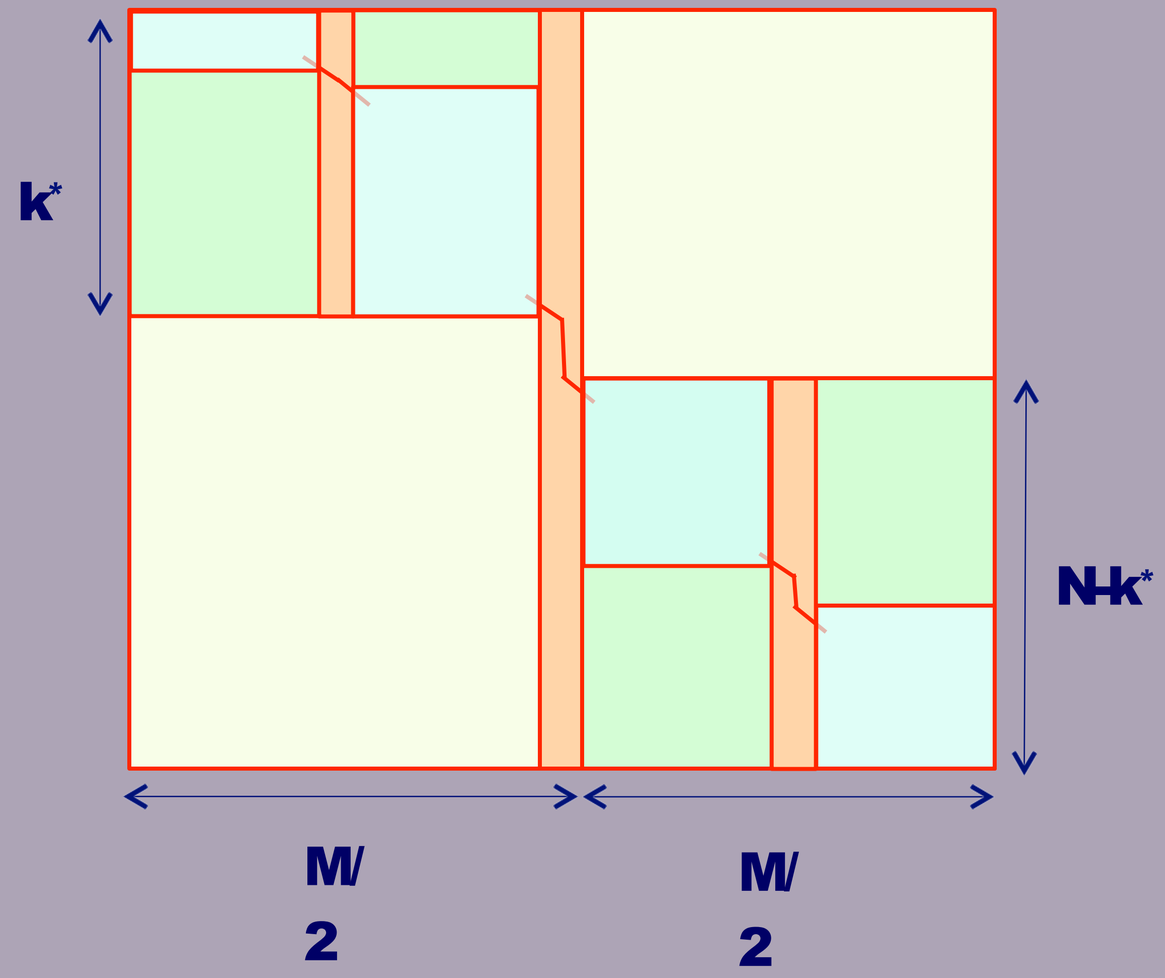


- Now, we can find k^* maximizing $F(M/2, k) + F^r(M/2, N-k)$
- Also, we can trace the path exiting column M/2 from k^*

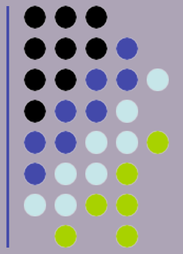


Linear-space alignment

- Iterate this procedure to the left and right!



Linear-space alignment



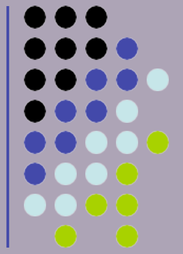
Hirschberg's Linear-space algorithm:

MEMALIGN(l, l', r, r'): (aligns $x_l \ldots x_{l'}$ with $y_r \ldots y_{r'}$)

1. Let h = $\lceil (l'-l)/2 \rceil$
2. Find (in Time $O((l' - l) \times (r' - r))$, Space $O(r' - r)$)
   the optimal path, L_h, entering column h – 1, exiting column h
   Let k_1 = pos'n at column h – 2 where L_h enters
   k_2 = pos'n at column h + 1 where L_h exits
3. MEMALIGN(l, h – 2, r, k_1)
4. Output L_h
5. MEMALIGN(h + 1, l', k_2, r')

Top level call: MEMALIGN(1, M, 1, N)

Linear-space alignment



Time, Space analysis of Hirschberg's algorithm:

To compute optimal path at middle column,

For box of size $M \times N$,

Space: 2N

Time: cMN, for some constant c

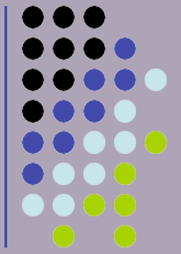
Then, left, right calls cost $c(M/2 \times k^* + M/2 \times (N - k^*)) = cMN/2$

All recursive calls cost

Total Time: $cMN + cMN/2 + cMN/4 + \ldots = 2cMN = O(MN)$

Total Space: O(N) for computation,

O(N + M) to store the optimal alignment



Heuristic Local Alignerers

1. The basic indexing & extension technique

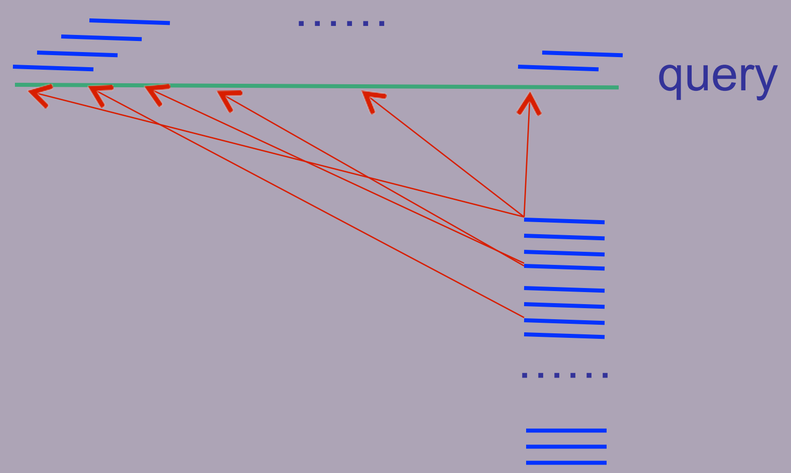
2. Indexing: techniques to improve sensitivity
   Pairs of Words, Patterns

3. Systems for local alignment

Indexing-based local alignment

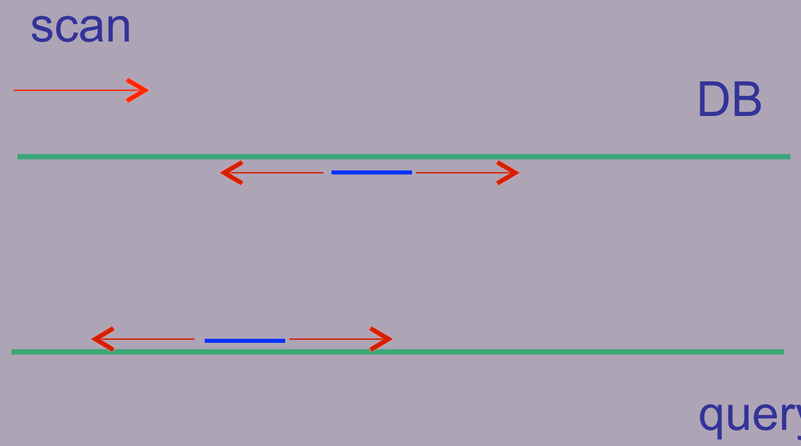
Dictionary:

All words of length k (~10)
Alignment initiated between words of alignment score ≥ T (typically T = k)



Alignment:

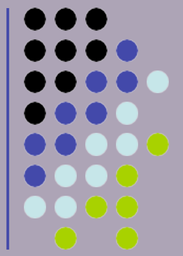
Ungapped extensions until score below statistical threshold



Output:

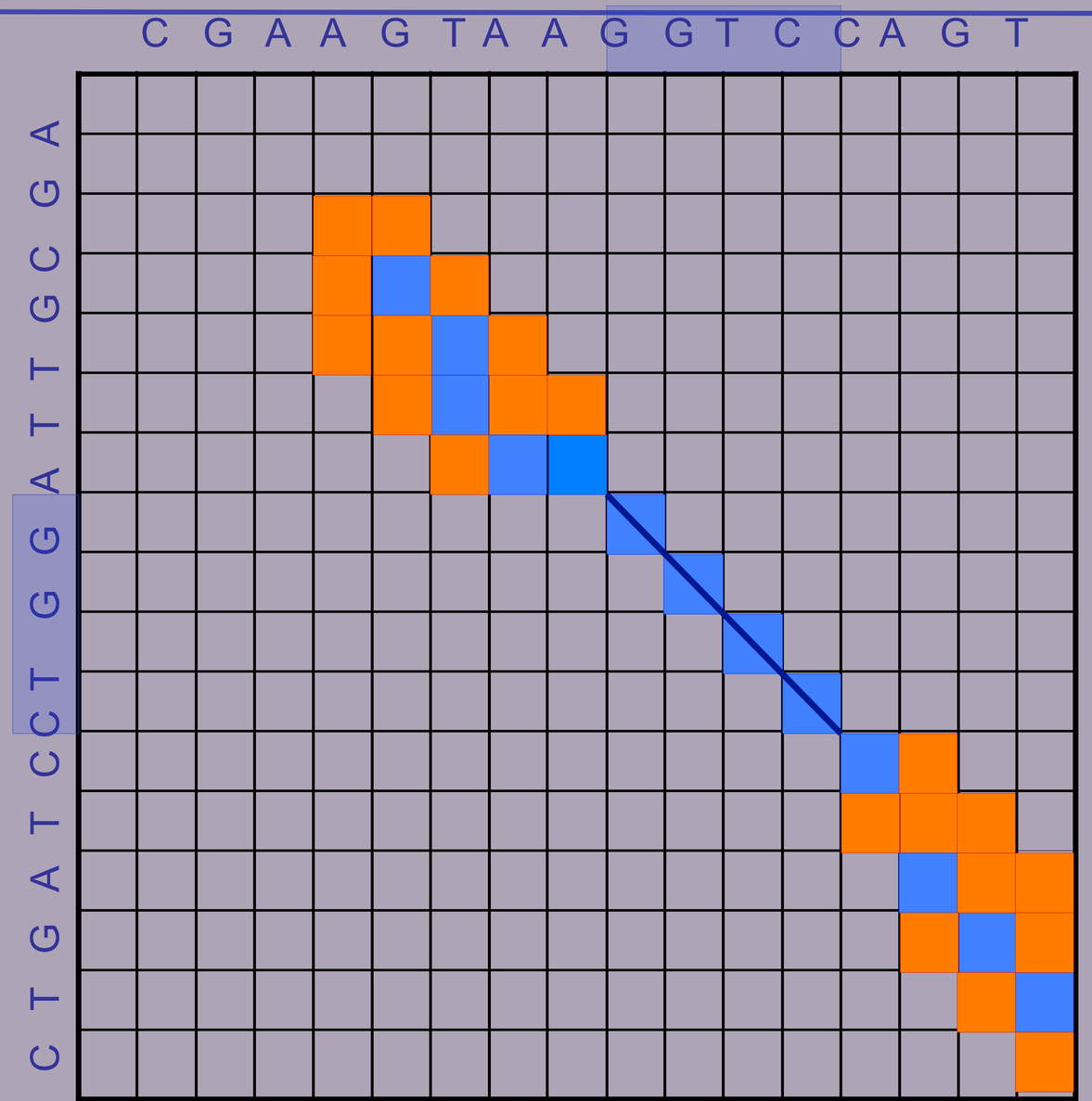
All local alignments with score > statistical threshold

Indexing-based local alignment—Extensions



Gapped extensions until threshold

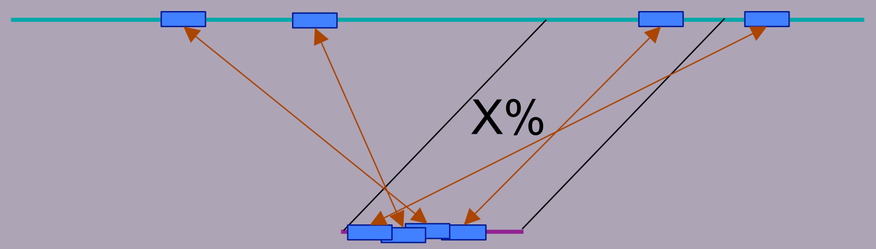
- Extensions with gaps until score < C below best score so far



Output:

```
GTAAGGTCCAGT
GTTAGGTC-AGT
```

Sensitivity-Speed Tradeoff



	long words (k = 15)	short words (k = 7)
Sensitivity		✓
Speed	✓	

Sens.

Speed

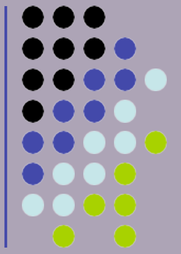
Table 3. Sensitivity and Specificity of Single Perfect Nucleotide K-mer Matches as a Search Criterion

	7	8	9	10	11	12	13	14
A. 81%	0.974	0.915	0.833	0.726	0.607	0.486	0.373	0.314
83%	0.988	0.953	0.897	0.815	0.711	0.595	0.478	0.415
85%	0.996	0.978	0.945	0.888	0.808	0.707	0.594	0.532
87%	0.999	0.992	0.975	0.942	0.888	0.811	0.714	0.659
89%	1.000	0.998	0.991	0.976	0.946	0.897	0.824	0.782
91%	1.000	1.000	0.998	0.993	0.981	0.956	0.912	0.886
93%	1.000	1.000	1.000	0.999	0.995	0.987	0.968	0.957
95%	1.000	1.000	1.000	1.000	0.999	0.998	0.994	0.991
97%	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.999
B. K	7	8	9	10	11	12	13	14
F	1.3e+07	2.9e+06	635783	143051	32512	7451	1719	399

(*A*) Columns are for K sizes of 7–14. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated from equation 3 assuming a homologous region of 100 bases. The larger the value of K, the fewer homologies are detected.
(*B*) K represents the size of the perfect match. F shows how many perfect matches of this size expected to occur by chance according to equation 4 in a genome of 3 billion bases using a query of 500 bases.

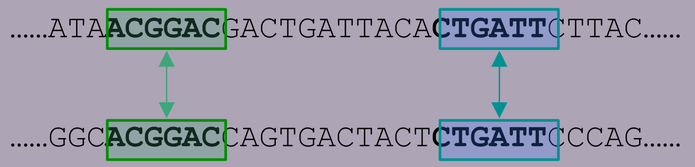
Kent WJ, Genome Research 2002

Sensitivity-Speed Tradeoff



Methods to improve sensitivity/speed

1. Using pairs of words



2. Using inexact words

3. Patterns—non consecutive positions

Measured improvement

Table 7. Sensitivity and Specificity of Multiple (2 and 3) Perfect Nucleotide K-mer Matches as a Search Criterion

	2,8	2,9	2,10	2,11	2,12	3,8	3,9	3,10	3,11	3,12
A. 81%	0.681	0.508	0.348	0.220	0.129	0.389	0.221	0.112	0.051	0.021
83%	0.790	0.638	0.475	0.326	0.208	0.529	0.339	0.193	0.099	0.045
85%	0.879	0.762	0.615	0.460	0.318	0.676	0.487	0.313	0.180	0.093
87%	0.942	0.866	0.752	0.611	0.461	0.809	0.649	0.470	0.305	0.177
89%	0.978	0.940	0.868	0.761	0.625	0.910	0.801	0.648	0.476	0.314
91%	0.994	0.980	0.947	0.884	0.787	0.969	0.914	0.815	0.673	0.505
93%	0.999	0.996	0.986	0.962	0.912	0.993	0.976	0.933	0.851	0.722
95%	1.000	1.000	0.998	0.993	0.979	0.999	0.997	0.987	0.961	0.902
97%	1.000	1.000	1.000	1.000	0.999	1.000	1.000	0.999	0.997	0.987

B. N,K	2,8	2,9	2,10	2,11	2,12	3,8	3,9	3,10	3,11	3,12
F	524	27	1.4	0.1	0.0	0.1	0.0	0.0	0.0	0.0

(*A*) Columns are for N sizes of 2 and 3 and K sizes of 8–12. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated by equation 10. (*B*) N and K represent the number and size of the near-perfect matches, respectively. F shows how many perfect clustered matches expected to occur by chance according to equation 14 in a translated genome of 3 billion bases using a query of 167 amino acids.

Table 5. Sensitivity and Specificity of Single Near-Perfect (One Mismatch Allowed) Nucleotide K-mer Matches as a Search Criterion

	12	13	14	15	16	17	18	19	20	21	22
A. 81%	0.945	0.880	0.831	0.721	0.657	0.526	0.465	0.408	0.356	0.255	0.218
83%	0.975	0.936	0.904	0.820	0.770	0.649	0.591	0.535	0.480	0.361	0.318
85%	0.991	0.971	0.954	0.900	0.865	0.767	0.719	0.669	0.619	0.490	0.445
87%	0.997	0.990	0.983	0.954	0.935	0.867	0.833	0.796	0.757	0.634	0.591
89%	1.000	0.997	0.995	0.984	0.976	0.939	0.920	0.897	0.872	0.775	0.741
91%	1.000	1.000	0.999	0.996	0.994	0.979	0.971	0.962	0.950	0.890	0.869
93%	1.000	1.000	1.000	0.999	0.999	0.996	0.994	0.991	0.988	0.963	0.954
95%	1.000	1.000	1.000	1.000	1.000	1.000	0.999	0.999	0.999	0.994	0.992
97%	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

B. K	12	13	14	15	16	17	18	19	20	21	22
F	275671	68775	17163	4284	1070	267	67	17	4.2	1.0	0.3

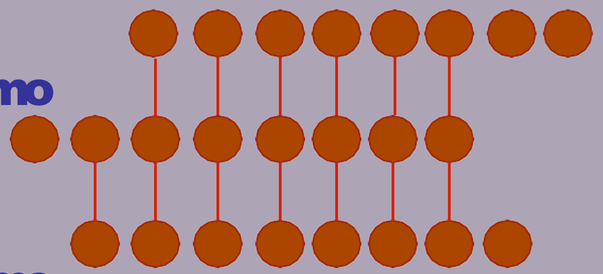
(*A*) Columns are for K sizes of 12–22. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated by equation 6 assuming a homologous region of 100 bases. (*B*) K represents the size of the near-perfect match. F shows how many perfect matches of this size expected to occur by chance accordi[illegible] billion bases using a query of 500 bases.

Kent WJ, Genome Research 2002

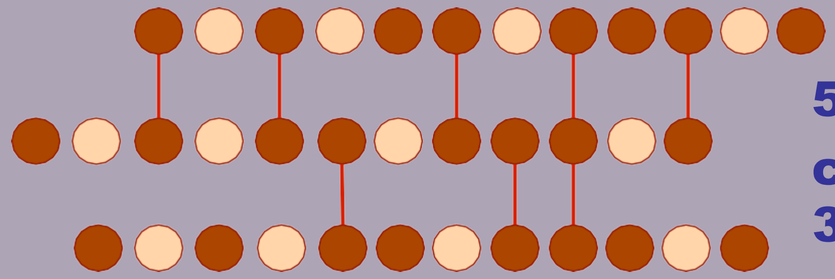
Non-consecutive words—Patterns

Patterns increase the likelihood of *at least one* match within a long conserved region

Consecutive Positions



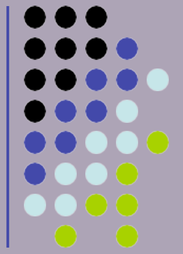
Non-Consecutive Positions



On a 100-long 70% conserved region:

	Consecutive	Non-consecutive
Expected # hits:	1.07	0.97
Prob[at least one hit]:	0.30	0.47

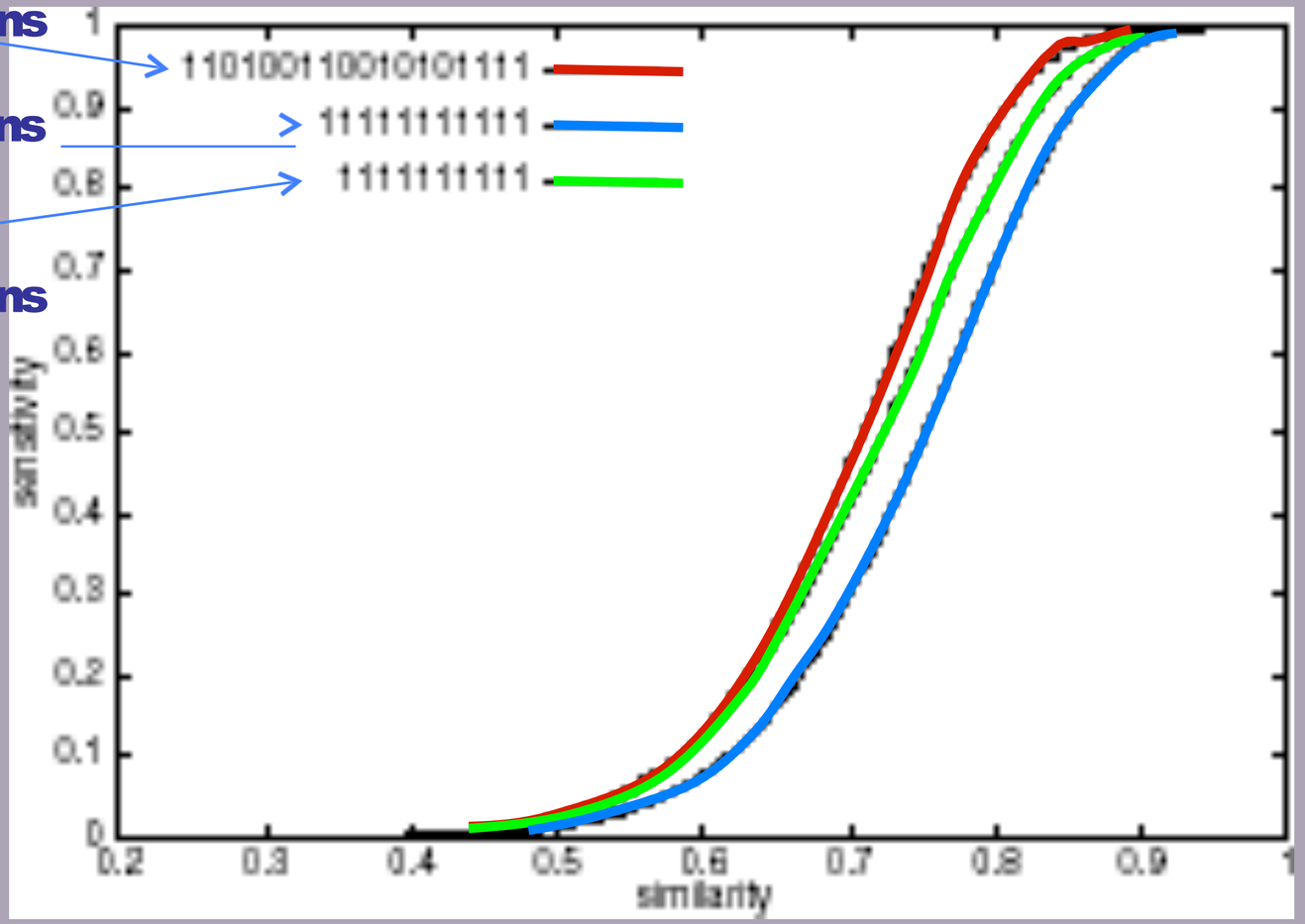
Advantage of Patterns



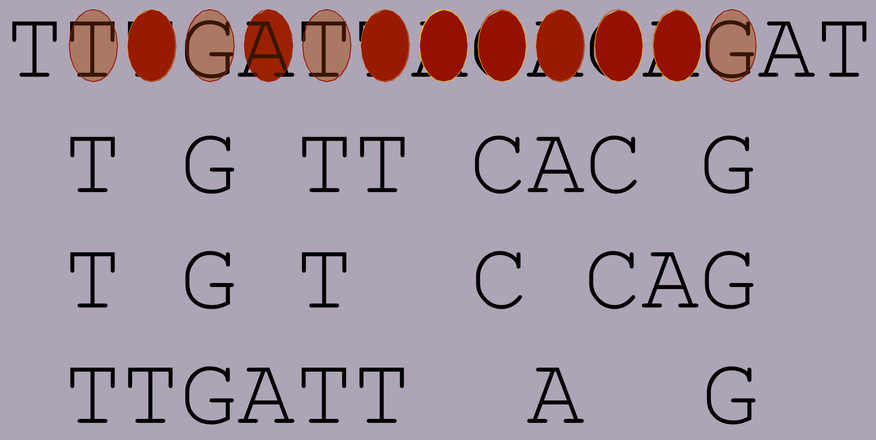
11 positions

11 positions

10 positions



Multiple patterns



How long does it take to search the query?

- K patterns

Seed	Pattern	Pr[detection]	Alignments Found	Time (s)
π_c	$\{0,1,2,3,4,5,6,7,8,9,10\}$	0.600	66419	15802
π_{c10}	$\{0,1,2,3,4,5,6,7,8,9\}$	0.707	73539	24129
π_{ph}	$\{0,1,2,4,7,9,12,13,15,16,17\}$	0.691	75518	16717
π_{N0}	$\{0,1,2,4,7,8,11,13,16,17,18\}$	0.683	75231	16225
π_{N5}	$\{0,1,2,3,5,6,7,10,12,13,14\}$	0.709	75547	16817
$\pi_1 + \pi_2$	$\{0,1,2,4,5,9,14,16,17,18,19,20\}+$ $\{0,1,2,3,4,6,7,8,10,11,12,13\}$	0.744	77211	22033

Buhler et al. RECOMB 2003
Sun & Buhler RECOMB 2004

Human Genome Resequencing

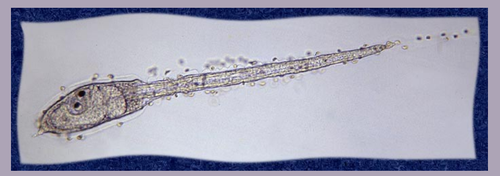
Which human did we sequence?

Answer one:

Answer two: “it doesn’t matter”

Polymorphism rate: number of letter changes between two different members of a species

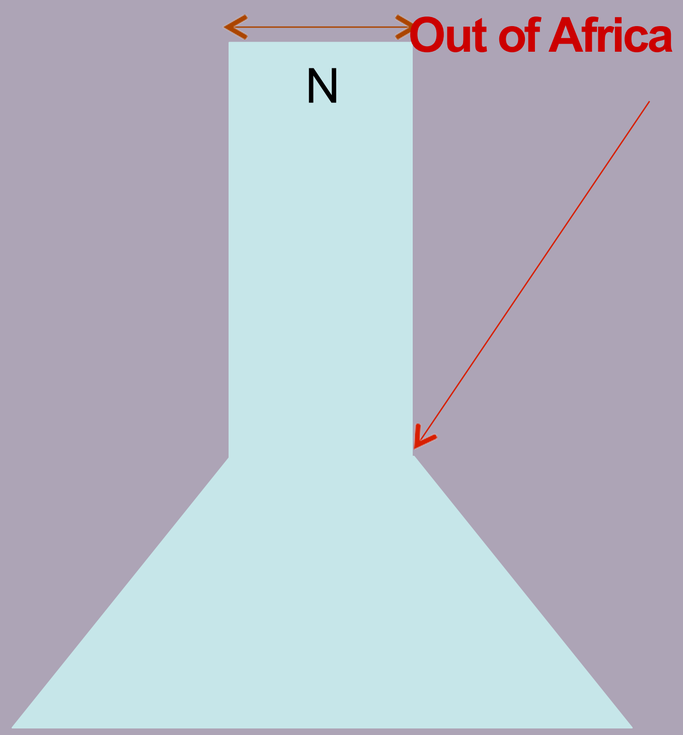
Humans: ~1/1,000



Other organisms have much higher polymorphism rates

- Population size!

Why humans are so similar



Heterozygosity: H

$H = 4Nu/(1 + 4Nu)$

$u \sim 10^{-8}$, $N \sim 10^{4}$

$\Rightarrow H \sim 4\times10^{-4}$

A small population that interbred reduced the genetic variation

Out of Africa ~ 40,000 years ago

DNA Sequencing

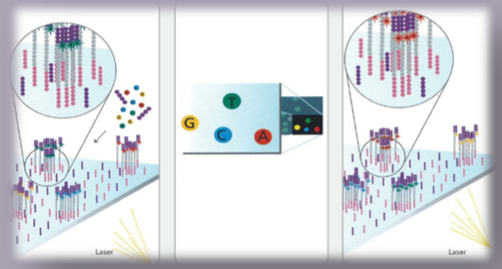
Goal:

Find the complete sequence of A, C, G, T's in DNA

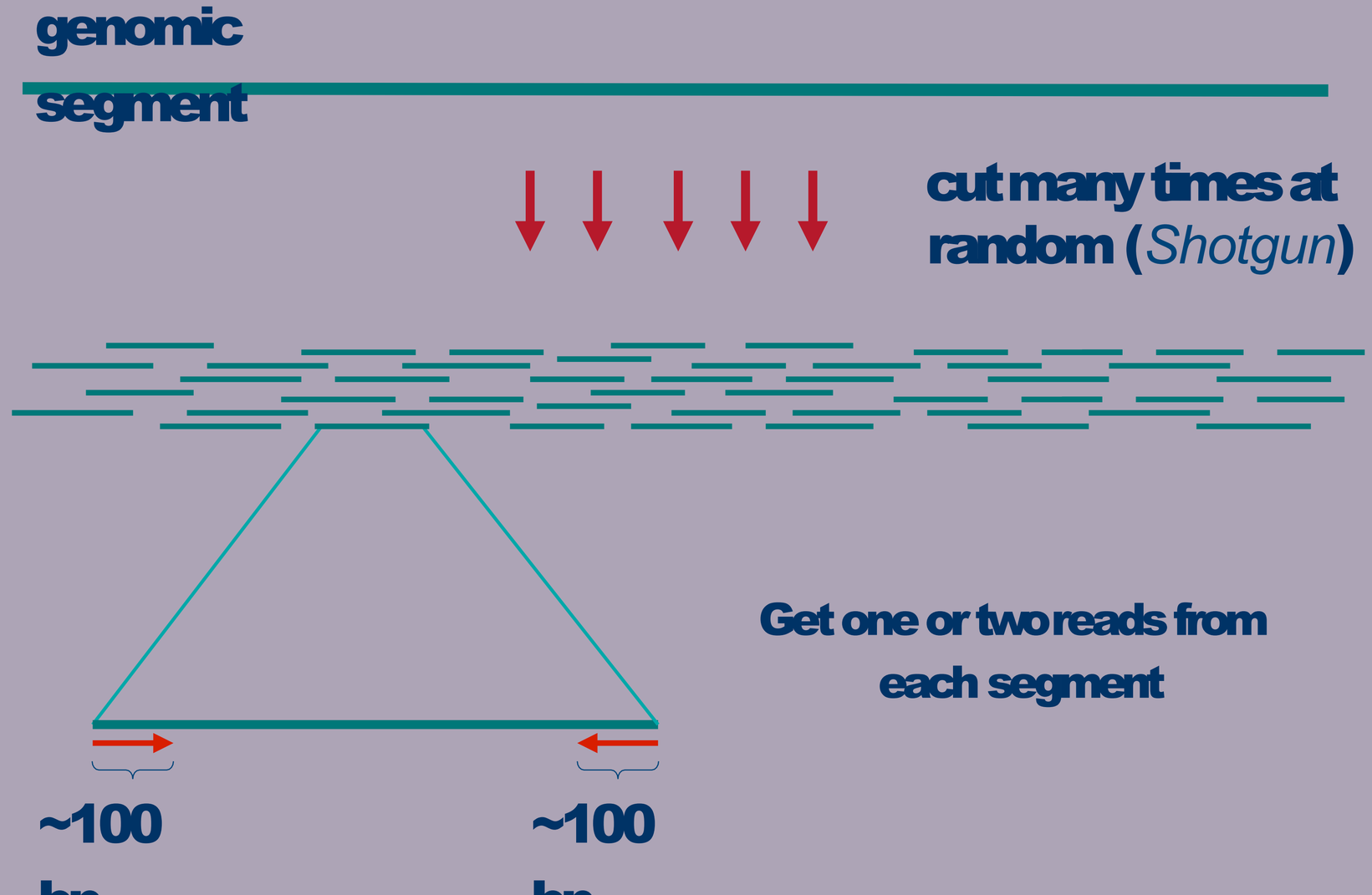
Challenge:

There is no machine that takes long DNA as an input, and gives the complete sequence as output

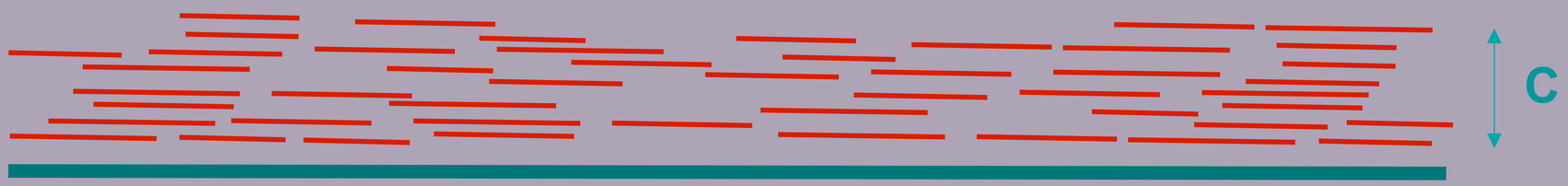
Can only sequence ~150 letters at a time



Method to sequence longer regions



Definition of Coverage



Length of genomic segment: **G**
Number of reads: **N**
Length of each read: **L**

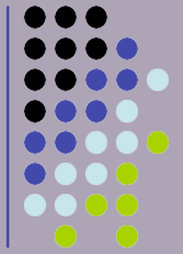
Definition: Coverage **C = N L / G**

How much coverage is enough?

Lander-Waterman model: **Prob[not covered bp] = e^{-C}**

Assuming uniform distribution of reads, C=10 results in 1 gapped region /1,000,000 nucleotides

Repeats



Bacterial genomes: 5%
Mammals: 50%

Repeat types:

- **Low-Complexity DNA** (e.g.ATATATATACATA…)
- **Microsatellite repeats** $(a_1 \ldots a_k)^N$ where k ~ 3-6
  (e.g. CAGCAGTAGCAGCACCAG)
- **Transposons**
  - **SINE** (Short Interspersed Nuclear Elements)
    e.g., ALU: ~300-long, 10^6 copies
  - **LINE** (Long Interspersed Nuclear Elements)
    ~4000-long, 200,000 copies
  - **LTR retroposons** (Long Terminal Repeats (~700 bp) at each end)
    cousins of HIV
- **Gene Families** genes duplicate & then diverge (paralogs)
- **Recent duplications** ~100,000-long, very similar copies

Two main assembly problems

- De Novo Assembly

- Resequencing

Human Genome Variation

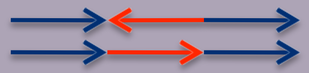
SNP

TGCTGAGA
TGCCGAGA

Novel Sequence

TGCTCGGAGA
TGC - - - GAGA

Inversion



Mobile Element or
Pseudogene Insertion

Translocation

Tandem Duplication

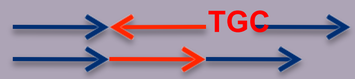
Microdeletion

TGC - - AGA
TGCCGAGA

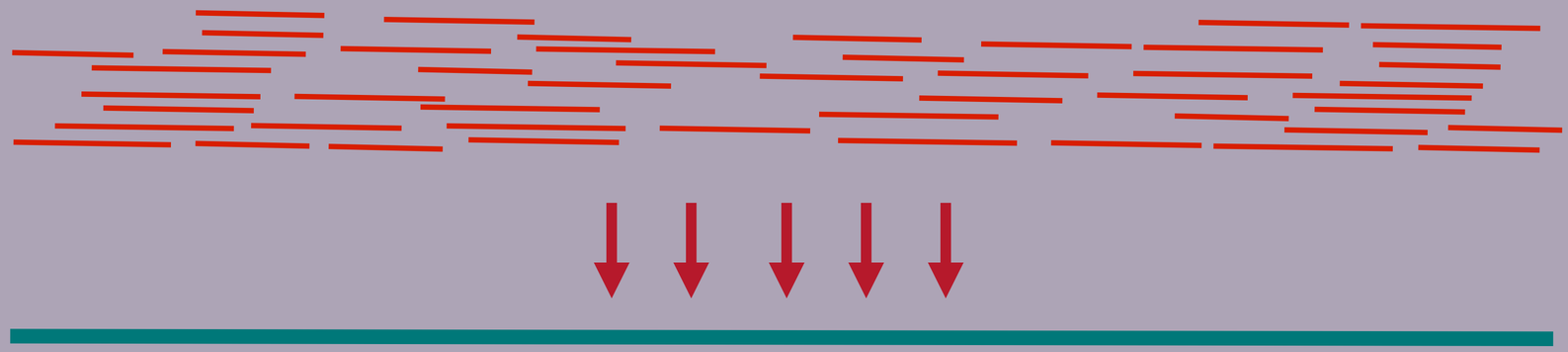
Transposition

Large Deletion

Novel Sequence
at Breakpoint



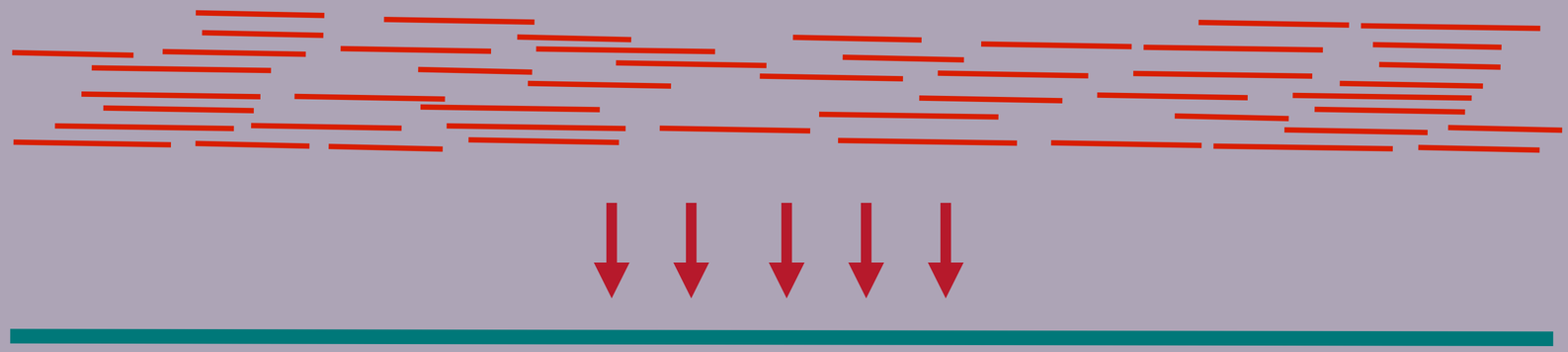
Read Mapping



```
      CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
    TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
  GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
......AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Want ultra fast, highly similar alignment
- Detection of genomic variation

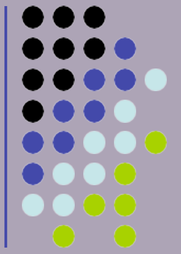
Read Mapping – Burrows-Wheeler Transform



```
      CATCGACCGAGCGCGATGCTAGCTAGGTGATCGT......
    TGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT...
  GCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATCGT
GTGCATGCCGCATCGACCGAGCGCGATGCTAGCTAGGTGATC
......AGGTGCATGCCGCATCGATCGAGCGCGATGCTAGCTAGCTGATCGT......
```

- Modern fast read aligners: BWT, Bowtie, SOAP
  - Based on ***Burrows-Wheeler transform***

Burrows-Wheeler Transform



ANA

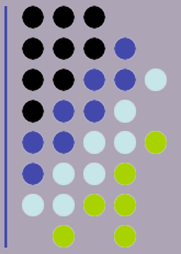
X = **BANANA**

BANANA
ANANA
NANA
ANA
NA
A

BANANA
ANANA
NANA
ANA
NA
A

suffixes of BANANA

Burrows-Wheeler Transform

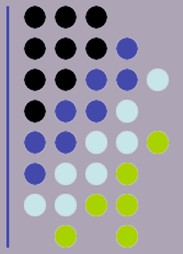


X = **BANANA$**

ANA

BANANA$	**BANANA$**
ANANA$	**ANA**NA$
NANA$	**NANA$**
ANA$	**ANA**$
NA$	**NA$**
A$	**A$**
$	**$**

Summary of BWT algorithm

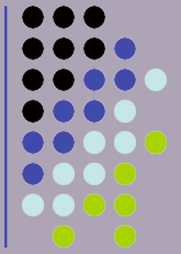


Suffix array of string X:

S(i) = j, where $X_j \ldots X_n$ is the j-th suffix lexicographically

- BWT follows immediately from suffix array
  - Suffix array construction possible in O(n), many good O(n log n) algorithms
- Reconstruct X from BWT(X) in time O(n)
- Search for all exact occurrences of W in time O(|W|)
- BWT(X) is easier to compress than X

Burrows-Wheeler Transform

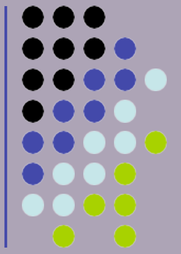


X = BANANA$

ANA

BANANA$	BANANA$	BANANA$
ANANA$B	ANANA$B	ANANA$B
NANA$BA	NANA$BA	NANA$BA
ANA$BAN	ANA$BAN	ANA$BAN
NA$BANA	NA$BANA	NA$BANA
A$BANAN	A$BANAN	A$BANAN
$BANANA	$BANANA	$BANANA

Burrows-Wheeler Transform

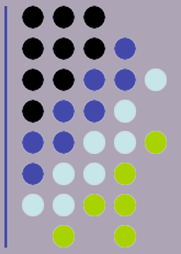


ANA

X = BANANA$

BANANA$	BANANA$	$BANANA
ANANA$B	ANANA$B	A$BANAN
NANA$BA	NANA$BA	ANA$BAN
ANA$BAN	ANA$BAN	ANANA$B
NA$BANA	NA$BANA	BANANA$
A$BANAN	A$BANAN	NA$BANA
$BANANA	$BANANA	NANA$BA

Burrows-Wheeler Transform

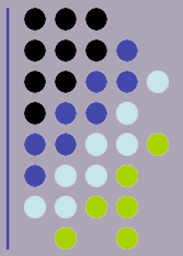


ANA

X = BANANA$

BANANA$	BANANA$	$BANANA
ANANA$B	ANANA$B	A$BANAN
NANA$BA	NANA$BA	ANA$BAN
ANA$BAN	ANA$BAN	ANANA$B
NA$BANA	NA$BANA	BANANA$
A$BANAN	A$BANAN	NA$BANA
$BANANA	$BANANA	NANA$BA

Burrows-Wheeler Transform



ANA

X = BANANA$

BANANA$
ANANA$B
NANA$BA
ANA$BAN
NA$BANA
A$BANAN

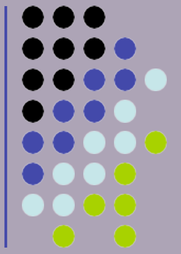
BANANA$
ANANA$B
NANA$BA
ANA$BAN
NA$BANA
A$BANAN

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

BWT matrix of string ‘BANANA’

BWT(BANANA) = ANNB$AA

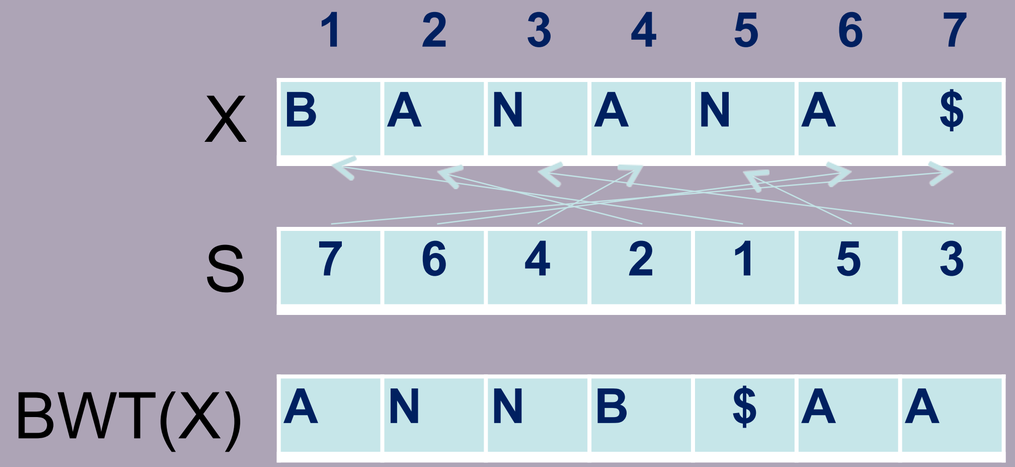
Suffix Arrays



$BANANA	1	$BANANA
A$BANAN	2	A$BANAN
ANA$BAN	3	ANA$BAN
ANANA$B	4	ANANA$B
BANANA$	5	BANANA$
NA$BANA	6	NA$BANA
NANA$BA	7	NANA$BA

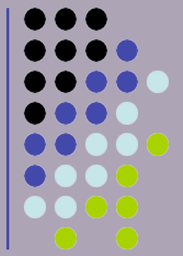
Suffixes are sorted in the BWT matrix

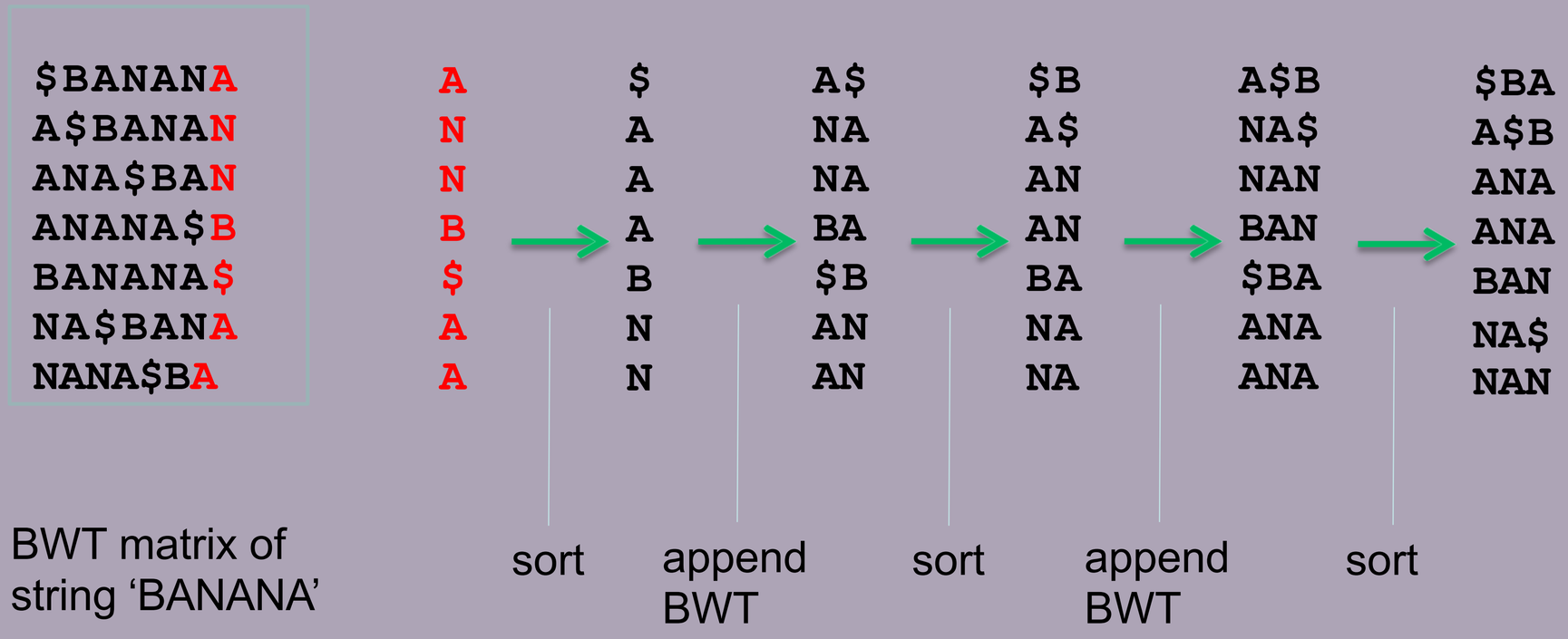
S(i) = j, where $X_j \ldots X_n$ is the i-th suffix lexicographically



BWT(X) constructed from S:
At each position, take the letter to the left of the one pointed by S

Reconstructing BANANA





BWT matrix of string ‘BANANA’

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

```
A $BANAN
N A$BANA
N ANA$BA
B ANANA$
$ BANANA
A NA$BAN
ANANA$B
```

Reconstructing BANANA - faster

$BANANA
A$BANAN
ANA$BAN
ANANA$B

BANANA$
NA$BANA
NANA$BA

BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character c in last column is the same text character as the i-th occurrence of c in the first column

A $BANAN
N A$BANA
N ANA$BA
BANANA$
$ BANANA
A NA$BAN
ANANA$B

A$BANAN
ANA$BAN
ANANA$B

Same words, same sorted order

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

Lemma. The i-th occurrence of character ‘a’ in last column is the same text character as the i-th occurrence of ‘a’ in the first column

LF(): Map the i-th occurrence of character ‘a’ in last column to the first column

LF(r): Let row r contain the i-th occurrence of ‘a’ in last column

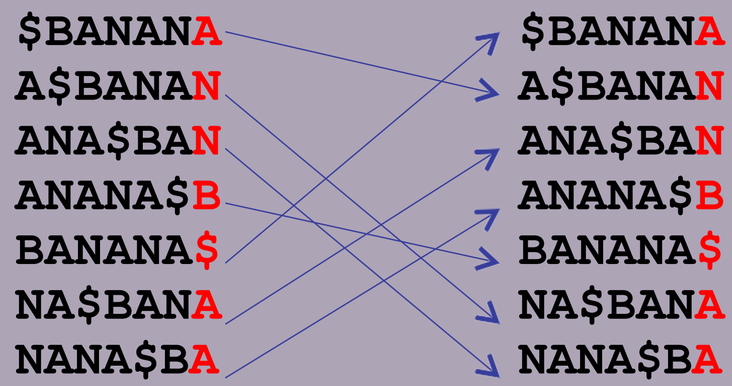
Then, LF(r) = r’; r’: i-th row starting with ‘a’

Reconstructing BANANA - faster

```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

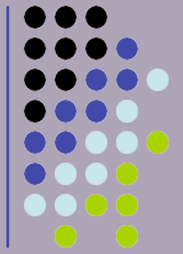
LF(r): Let row r be the i-th occurrence of ‘a’ in last column
Then, LF(r) = r’; r’: i-th row starting with ‘a’



LF[] = [2, 6, 7, 5, 1, 3, 4]

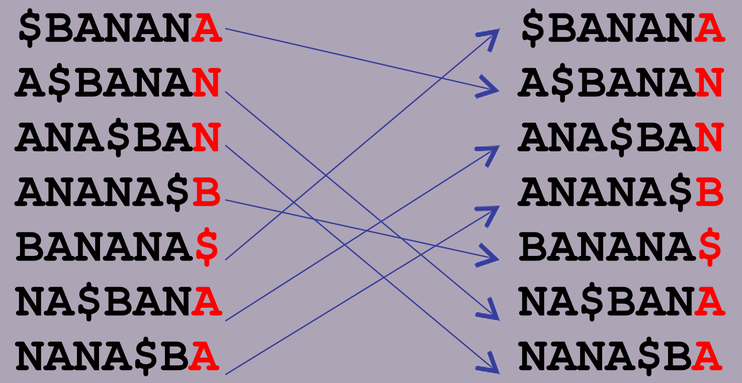
Row LF(r) is obtained by rotating row r one position to the right

Reconstructing BANANA - faster



```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’



LF[] = [2, 6, 7, 5, 1, 3, 4]

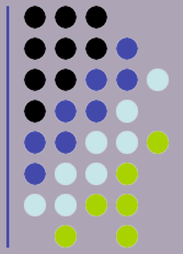
Computing LF() is easy:

Let C(a): # of characters smaller than ‘a’
Example: C($) = 0; C(A) = 1; C(B) = 4; C(N) = 5

Let row r end with the i-th occurrence of ‘a’ in last column

Then, LF(r) = C(a) + i (why?)

Reconstructing BANANA - faster



```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

	A	N	N	B	$	A	A
C()	1	5	5	4	0	1	1
index i	1	1	2	1	1	2	3
LF()	2	6	7	5	1	3	4

C() copied for convenience

indicating this is i-th occurrence of ‘c’

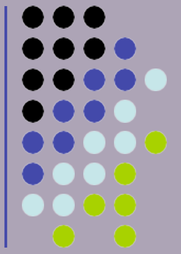
LF() = C() + i

Reconstruct BANANA:

```
S := “”; r := 1; c := BWT[r];
UNTIL c = ‘$’ {
    S := cS;
    r := LF(r);
    c := BWT(r); }
```

Credit: Ben Langmead thesis

Searching for ANA



```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

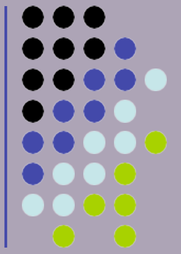
L(W): lowest index in BWT matrix where W is prefix
U(W): highest index in BWT matrix where W is prefix

Example:
L(“NA”) = 6
U(“NA”) = 7

Lemma (prove as exercise)
L(aW) = C(a) + i +1,
where i = # ‘a’s up to L(W) – 1 in BWT(X)
U(aW) = C(a) + j,
where j = # ‘a’s up to U(W) in BWT(X)

Example:
L(“ANA”) = C(‘A’) + # ‘A’s up to (L(“NA”) – 1) + 1
= 1 + (# ‘A’s up to 5) + 1
= 1 + 1 + 1 = 3
U(“ANA”) = 1 + # ‘A’s up to U(“NA”) = 1 + 3 = 4

Searching for ANA



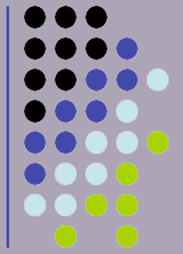
```
$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA
```

BWT matrix of string ‘BANANA’

```
Let
LFC(r, a) = C(a) + i, where i = #’a’s up to r in BWT
ExactMatch(W[1…k]) {

    a := W[k];

  low := C(a) +1;
  high := C(a+1);       // a+1: lexicographically next char
  i := k – 1;
  while (low <= high && i >= 1)
          {  a = W[i];
      low = LFC(low – 1, a) + 1;
      high = LFC(high, a);  i :=
      i – 1; }
  return (low, high);
}
```

Credit: Ben Langmead thesis

Summary of BWT algorithm



Suffix array of string X:

S(i) = j, where $X_j \ldots X_n$ is the j-th suffix lexicographically

- BWT follows immediately from suffix array
  - Suffix array construction possible in O(n), many good O(n log n) algorithms
- Reconstruct X from BWT(X) in time O(n)
- Search for all exact occurrences of W in time O(|W|)
- BWT(X) is easier to compress than X

BWT Index Construction

Reference Sequence Construction

BWT Construction

BWT-auxiliary Structure Construction (C & O arrays) and Compression

.bwt

Reference

TTATTT...ATGTGCCTTTGAAA...GTTAAACCT...AAATT.....AATTT...AGGTTTAAAC..TTTCAAAGGCACAT..AAATAA$

Forward *Reverse Complement*

BWT

ACGTTA..TTCTGAATGTGACC...TCCAGACGA...CCATT.....AGTTC...CGGATT AGAT...AAGTACCGTGTGAT...CCAGAT

Compressed BWT (4 bases/byte)

TTATTT..ATGTGCCTT.....$GTTGGTTAATAA

C-array

S:	0
T:	55000
C:	1044814
G:	7814189
A:	1

O-array

	T	G	C	A
0:	1	0	0	0
1:	2	0	0	0
2:	.....			
3:	.....			
..	.....			
|G|-1:	.....			

SA

0:	|G|-1
1:	64
2:	144814
3:	781414689
..:	...
|G|-1:	1484

Memory Consumption

For a genome of length n:

-- occurrence array O(.,.) needs 4nlogn bits

→ sampling: store only O(.,k) for e.g. k = 128

→ use BWT to compute missing counts

-- suffix array SA(.) needs nlogn bits

→ sampling: store SA(k) for e.g. k = 32

→ use inverse compressed suffix array

BWA Inexact Matching

Allow up to *n* mismatches/gaps.

Backwards-search extension:
Given read W, keep track of multiple possible partial alignments of W

Partial alignment 4-tuple: (i, z, L , U)

$I \leftarrow \emptyset$
$I \leftarrow I \cup \text{INEXRECUR}(W, i-1, z-1, k, l)$
for each $b \in \{A, C, G, T\}$ **do**
  $k \leftarrow C(b) + O(b, k-1) + 1$
  $l \leftarrow C(b) + O(b, l)$
  if $k \leq l$ **then**
    $I \leftarrow I \cup \text{INEXRECUR}(W, i, z-1, k, l)$
    if $b = W[i]$ **then**
      $I \leftarrow I \cup \text{INEXRECUR}(W, i-1, z, k, l)$
    else
      $I \leftarrow I \cup \text{INEXRECUR}(W, i-1, z-1, k, l)$

BWA Inexact Matching

W = ACTGTGT

Partial alignment 4-tuple: (i = 4, z = 3, L , U)

Recursive step:

A	C	T	G	gap-ref	gap-read
AGT	CGT	TGT	GGT	ǂGT	*GT
z-1	z-1	z	z-1	z-1	z-1
i-1	i-1	i-1	i-1	i-1	i
L^AU^A	L^CU^C	L^TU^T	L^GU^G	LU	L^AU^A L^CU^C L^TU^T L^GU^G
…GAGT	…GCGT	…GTGT	…GGGT	…G-GT	…GT[A/C/T/G]GT
…GTGT	…GTGT	…GTGT	…GTGT	…GTGT	…GT - GT

L^A=C(A) + O(A, L-1) + 1

U^A=C(A) + O(A, L)

```
I ← ∅
I ← I ∪ InexRecur(W, i−1, z−1, k, l)
for each b ∈ {A, C, G, T} do
    k ← C(b) + O(b, k−1) + 1
    l ← C(b) + O(b, l)
    if k ≤ l then
        I ← I ∪ InexRecur(W, i, z−1, k, l)
        if b = W[i] then
            I ← I ∪ InexRecur(W, i−1, z, k, l)
        else
            I ← I ∪ InexRecur(W, i−1, z−1, k, l)
```

BWA Heuristics

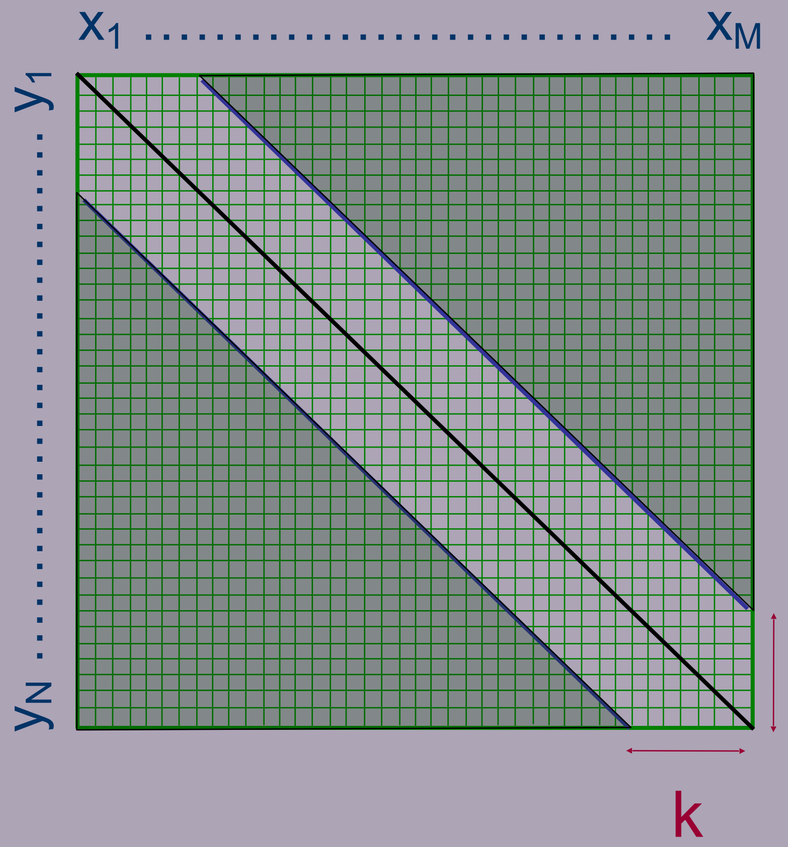
- Lower bound array *D*, where *D(i)* := LB on number of differences of exactly matching R[0,i] with the reference (can be computed in O(|R|) time → check $n < D(i)$ instead of $n < 0$)
- Process best partial alignments first: use a *min*-priority heap to store alignment entries (instead of recursion)
- Prune out alignments considered sub-optimal (although they might have fewer than *n* differences):
  dynamically adjust search parameters (e.g. *n*):
  - (1) stop if # top hits exceeds a threshold (=30),
  - (2) set $n = nbest + 1$, where *nbest* is the # of differences in top hit
- Seeding: limit the number of differences in the *seed* sequence (first *k* bp)
- Disallow indels at the ends of the read

General and affine gap penalties for edit distance

```
from pprint import pprint
from itertools import permutations
import sys
import fasta_parser
import alignment

if len(sys.argv) == 2:
    filename = sys.argv[1]
else:
    sys.exit("Usage: " + sys.argv[0] + " [fasta_filename]")

reads   = []
names   = []

with open(filename) as fp:
    for (name, seq) in fasta_parser.read_fasta(fp):
        names.append(name)
        reads.append(seq)

perms = permutations(reads, 2)

for perm in perms:
    print("Sequence1")
    print("Name: " + names[reads.index(perm[0])])
    print("Sequence: " + perm[0])
    print("Sequence2")
    print("Name: " + names[reads.index(perm[1])])
    print("Sequence: " + perm[1])
    [X, Y, M] = alignment.distance_matrix(perm[0], perm[1])
    [str1, str2] = alignment.backtrace(perm[0], perm[1], X, Y, M)
    print("-=Alignment=-")
    print(str1)
    print(str2)
    print("\n")
```

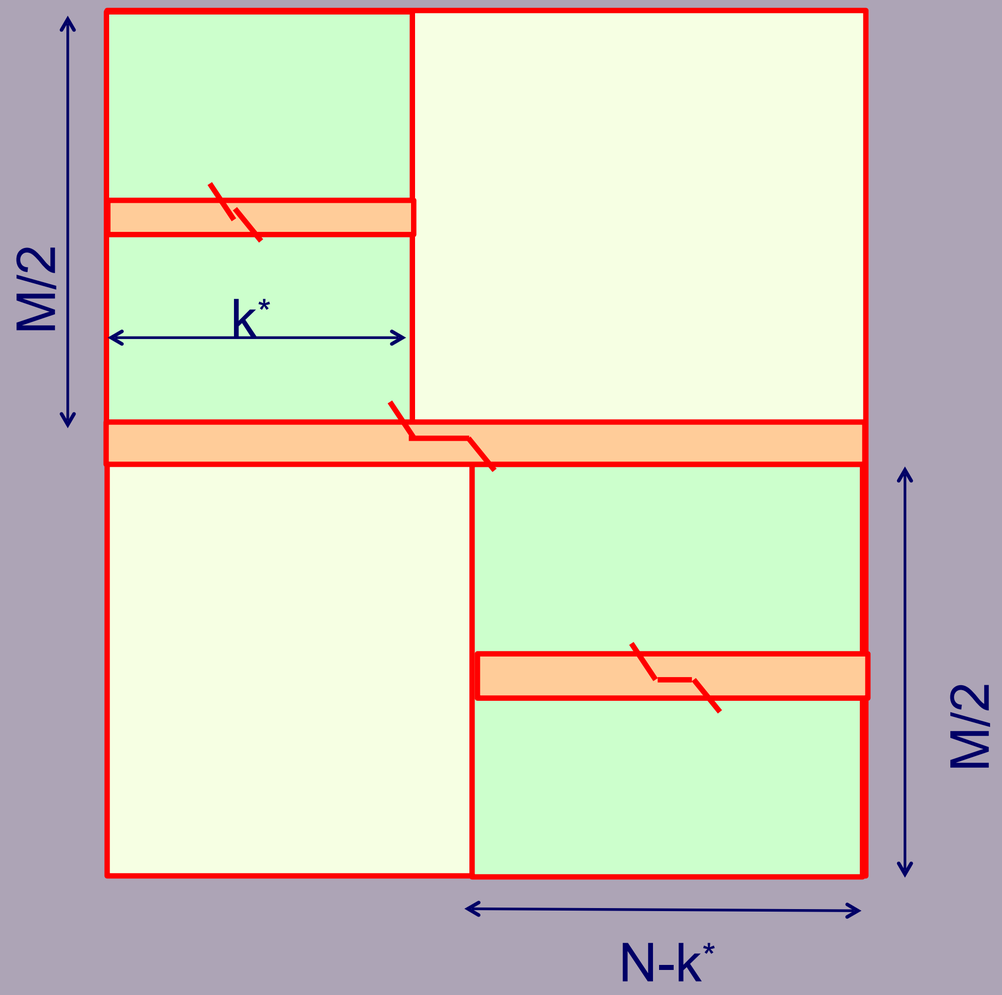
The affine gap penalty is analogous to paying a cable bill in which you need to pay a one-time installation fee (a) and get the first month of TV free, followed by paying by a monthly charge (b).

Bounded Dynamic Programming



- O(kM) time
- O(kN) memory

Linear-space alignment



- O(M+N) memory
- 2MN time

What’s a better alignment?

```
GACGCCGAACG
|||||   |||
GACGC---ACG
```

```
GACGCCGAACG
||||  |  |  ||
GACG-C-A-CG
```

Score = 8 x m – 3 x d

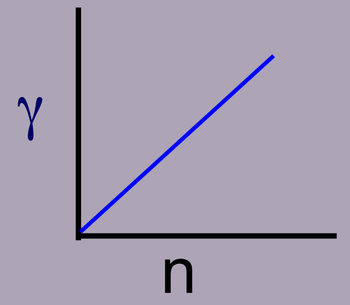
Score = 8 x m – 3 x d

However, gaps usually occur in bunches.

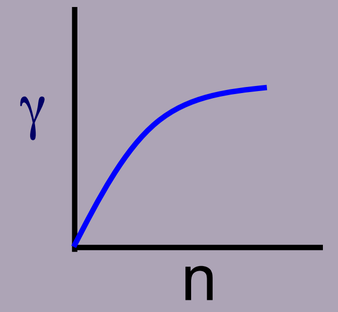
- During evolution, chunks of DNA may be lost entirely
- Aligning genomic sequences vs. cDNAs (reverse complimentary to mRNAs)

Model gaps more accurately

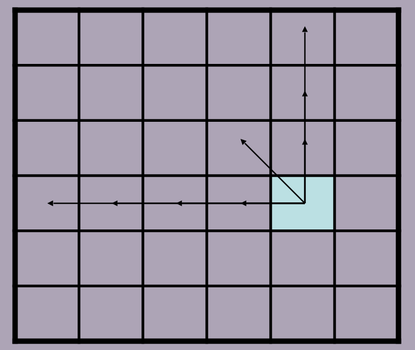
- Current model:
  - Gap of length n incurs penalty n×d



- General:
  - Convex function
  - E.g. $\gamma(n) = c * \text{sqrt}(n)$



General gap dynamic programming



Initialization: same

Iteration:

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ \max_{k=0\ldots i-1} F(k,j) - \gamma(i-k) \\ \max_{k=0\ldots j-1} F(i,k) - \gamma(j-k) \end{cases}$$

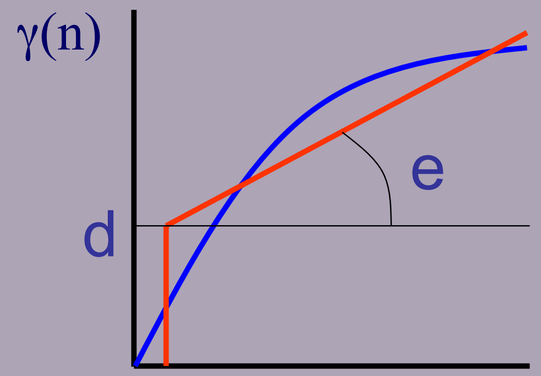
Termination: same

Running Time: O((M+N)MN) (cubic)
Space: O(NM) (linear-space algorithm not applicable)

Compromise: **affine gaps**

$\gamma(n) = d + (n - 1) \times e$

gap open	gap extension



Match: 2

Gap open: -5

Gap extension: -1

```
GACGCCGAACG
|||||   |||
GACGC---ACG
```

8x2-5-2 = 9

```
GACGCCGAACG
||||  |  |  ||
GACG-C-A-CG
```

8x2-3x5 = 1

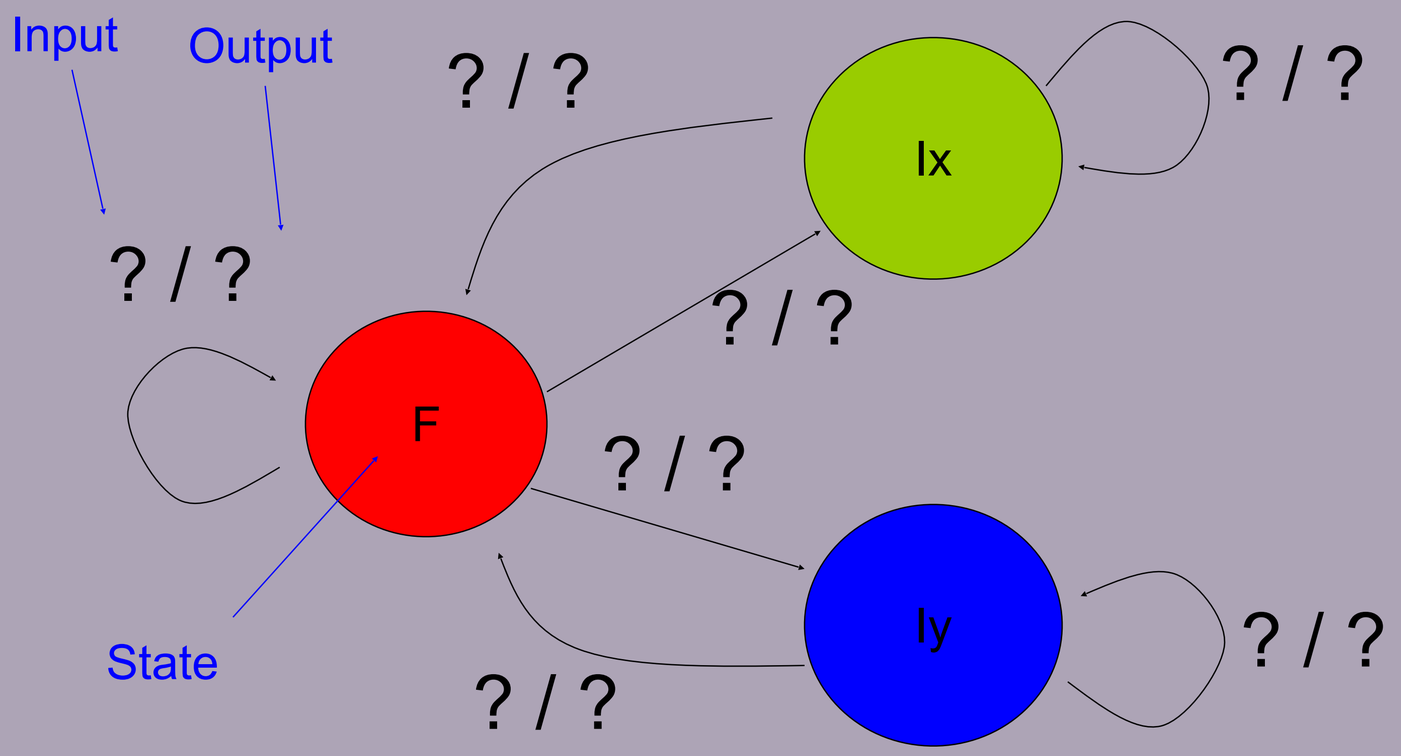
We want to find the optimal alignment with affine gap penalty in

- O(MN) time
- O(MN) or better O(M+N) memory

Allowing affine gap penalties

- Still three cases
  - x_i aligned with y_j
  - X_i aligns to a gap
    - Are we continuing a gap in x? (if no, start is more expensive)
  - Y_j aligns to a gap
    - Are we continuing a gap in y? (if no, start is more expensive)
- We can use a finite state machine to represent the three cases as three states
  - The machine has two heads, reading the chars on the two strings separately
  - At every step, each head reads 0 or 1 char from each sequence
  - Depending on what it reads, goes to a different state, and produces different scores

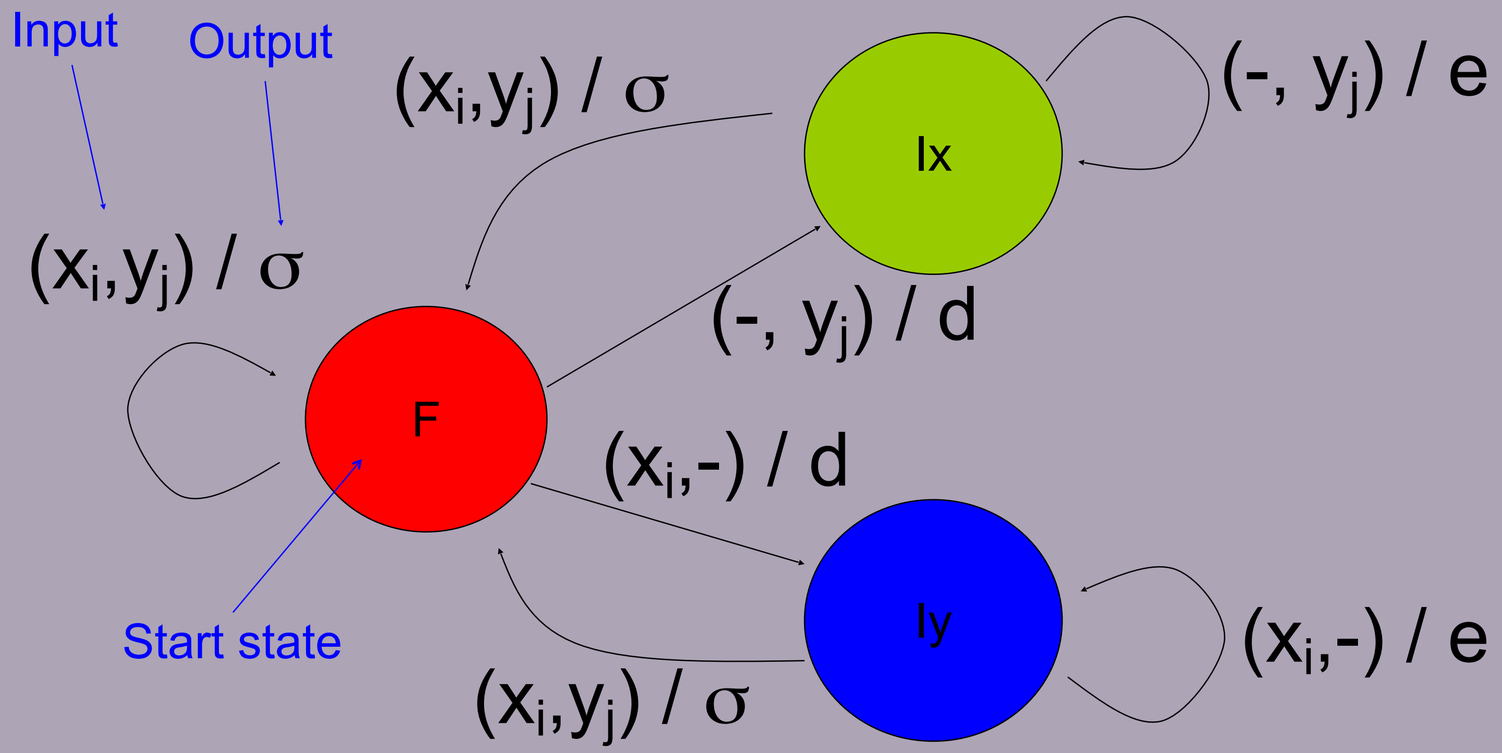
Finite State Machine



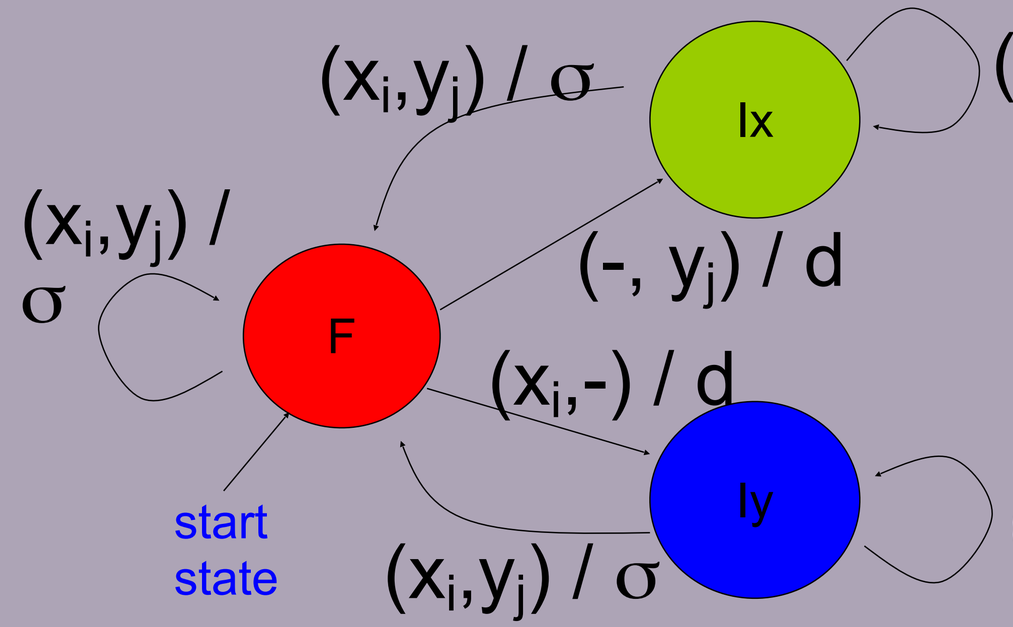
F: have just read 1 char from each seq (x_i aligned to y_j)

Ix: have read 0 char from x. (y_j aligned to a gap)

Iy: have read 0 char from y (x_i aligned to a gap)



Current state	Input	Output	Next state
F	(x_i,y_j)	σ	F
F	$(-,y_j)$	d	Ix
F	$(x_i,-)$	d	Iy
Ix	$(-,y_j)$	e	Ix
…	…	…	…



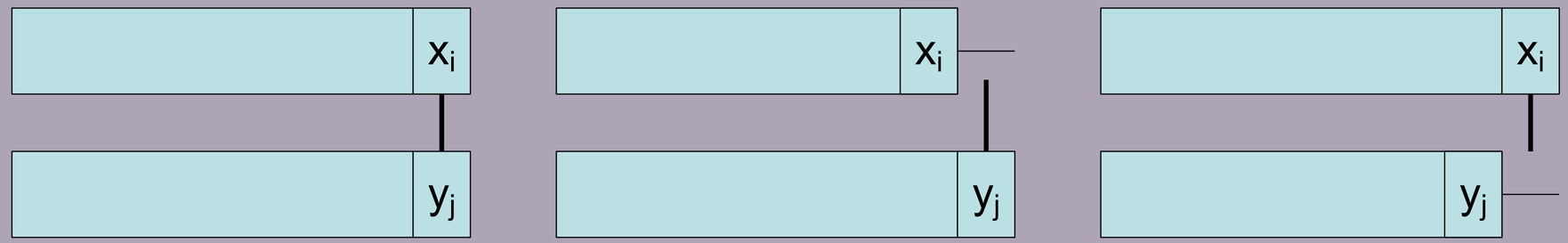
	F-F-F-F	F-I_y-F-F-I_x	F-F-I_y-F-I_x
AAC	AAC	AAC–	AAC–
ACT	| | |	| |	| |
	ACT	–ACT	A–CT

Given a pair of sequences, an alignment (not necessarily optimal) corresponds to a state path in the FSM.

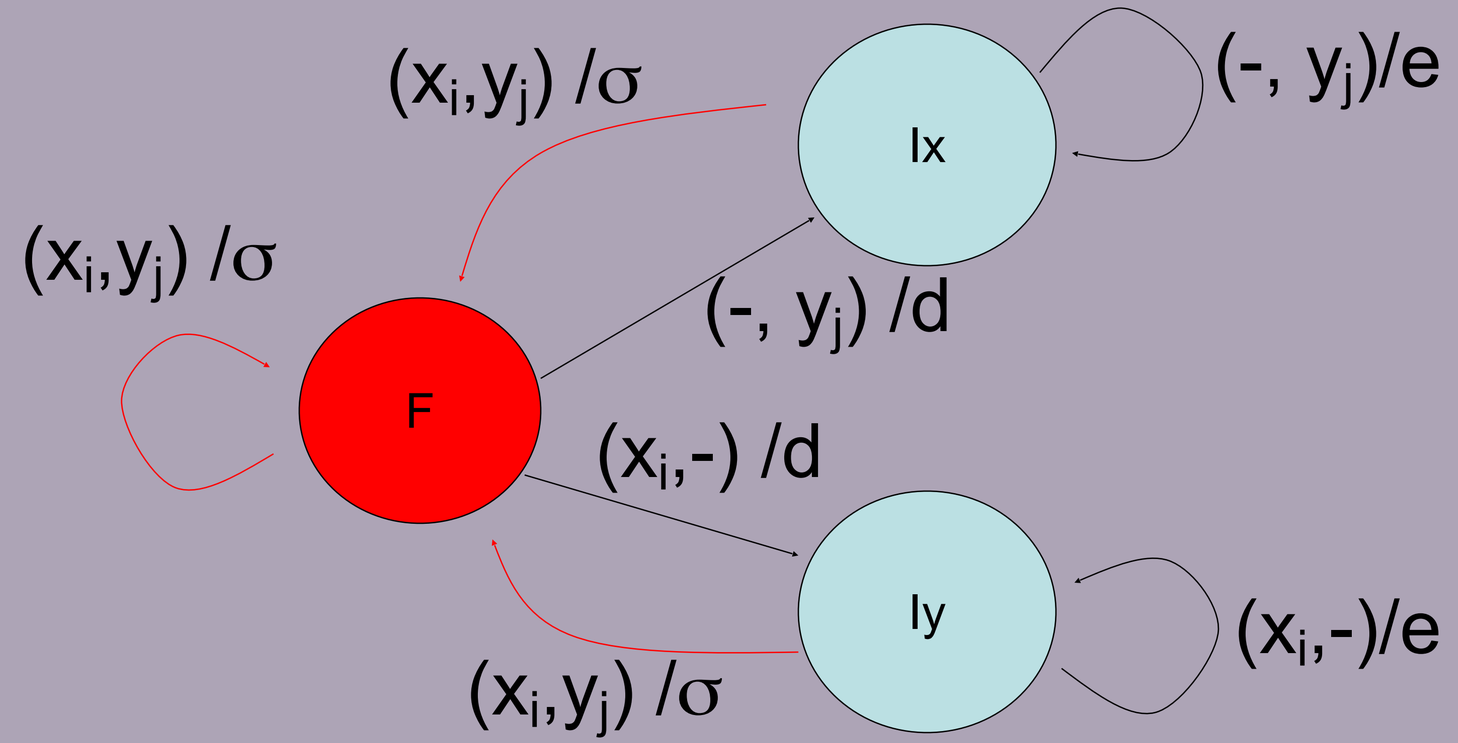
Optimal alignment: find a state path to read the two sequences such that the total output score is the highest

Dynamic programming

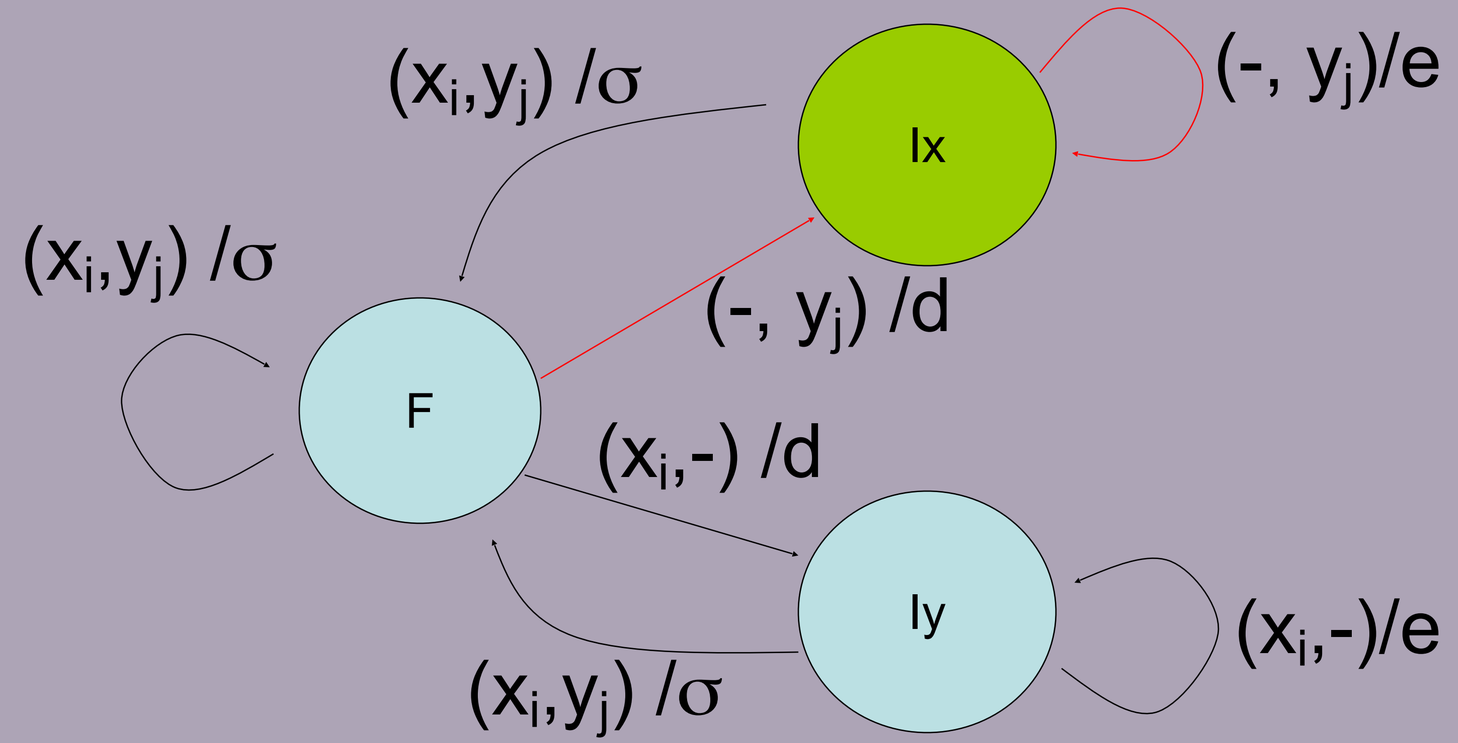
- We encode this information in three different matrices
- For each element (i,j) we use three variables
  - F(i,j): best alignment (score) of $x_1..x_i$ & $y_1..y_j$ if x_i aligns to y_j
  - I_x(i,j): best alignment of $x_1..x_i$ & $y_1..y_j$ if y_j aligns to gap
  - I_y(i,j): best alignment of $x_1..x_i$ & $y_1..y_j$ if x_i aligns to gap

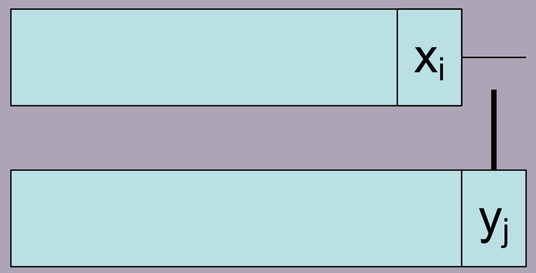


F(i, j) Ix(i, j) Iy(i, j)



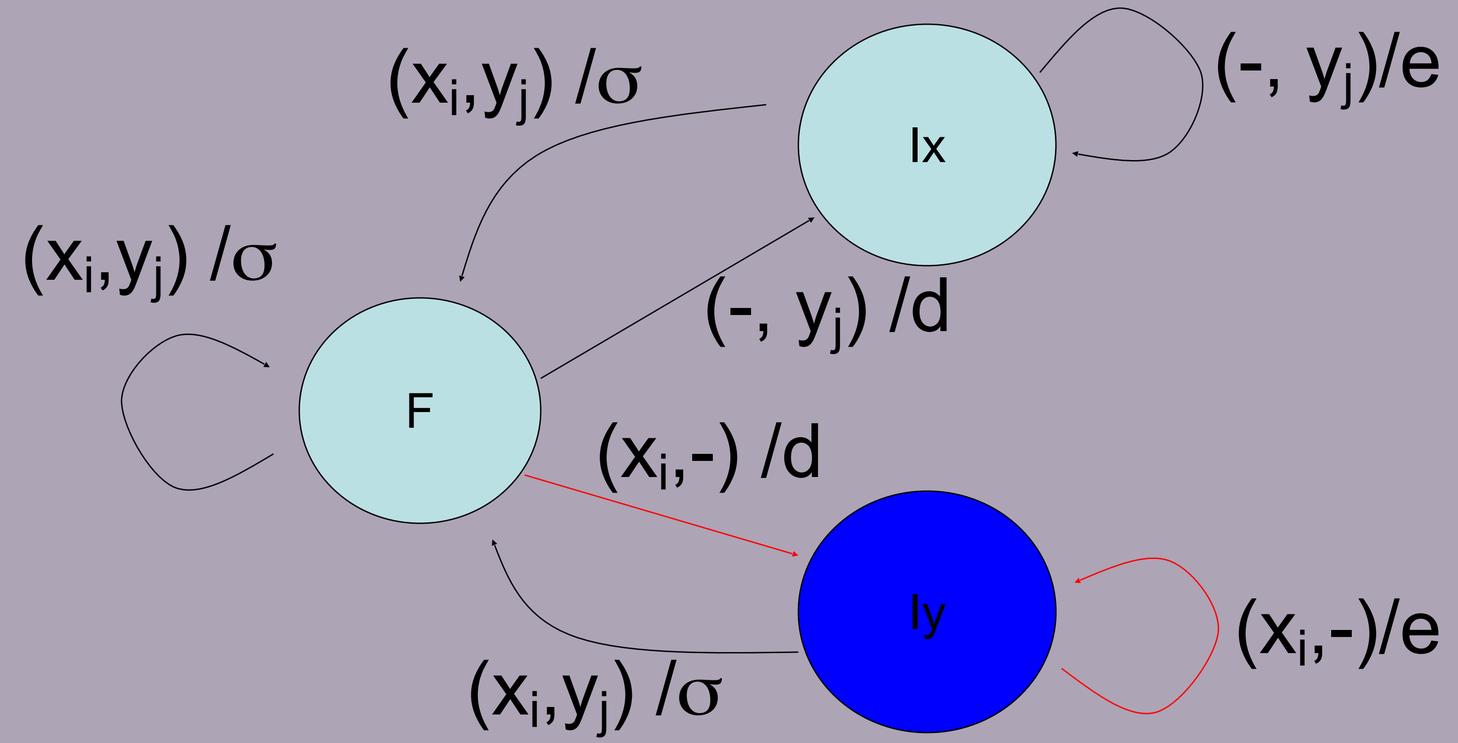
$$F(i, j) = \max \begin{cases} F(i-1, j-1) + \sigma(x_i, y_j) \\ Ix(i-1, j-1) + \sigma(x_i, y_j) \\ Iy(i-1, j-1) + \sigma(x_i, y_j) \end{cases}$$

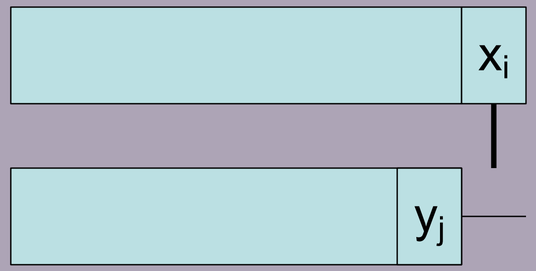




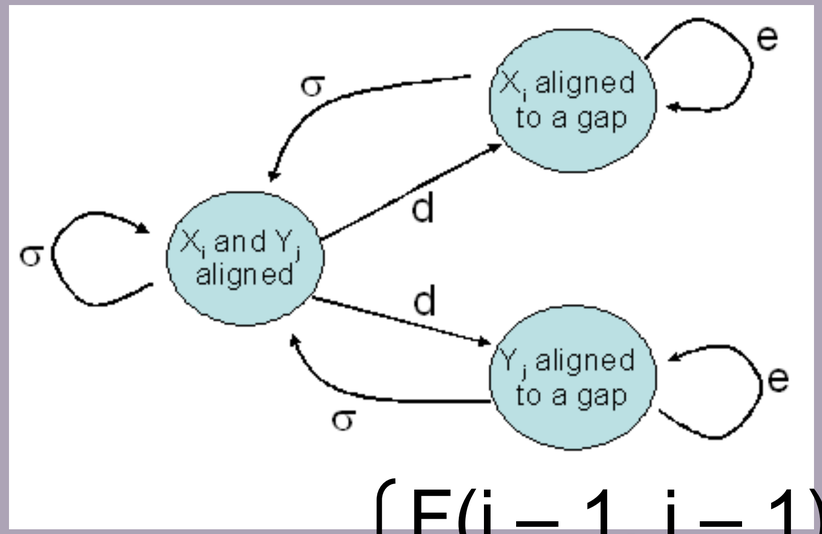
Ix(i, j)

$$Ix(i,j) = \max \begin{cases} F(i, j-1) + d \\ Ix(i, j-1) + e \end{cases}$$





$$Iy(i,j) = \max \begin{cases} F(i-1,j) + d \\ Iy(i-1,j) + e \end{cases}$$

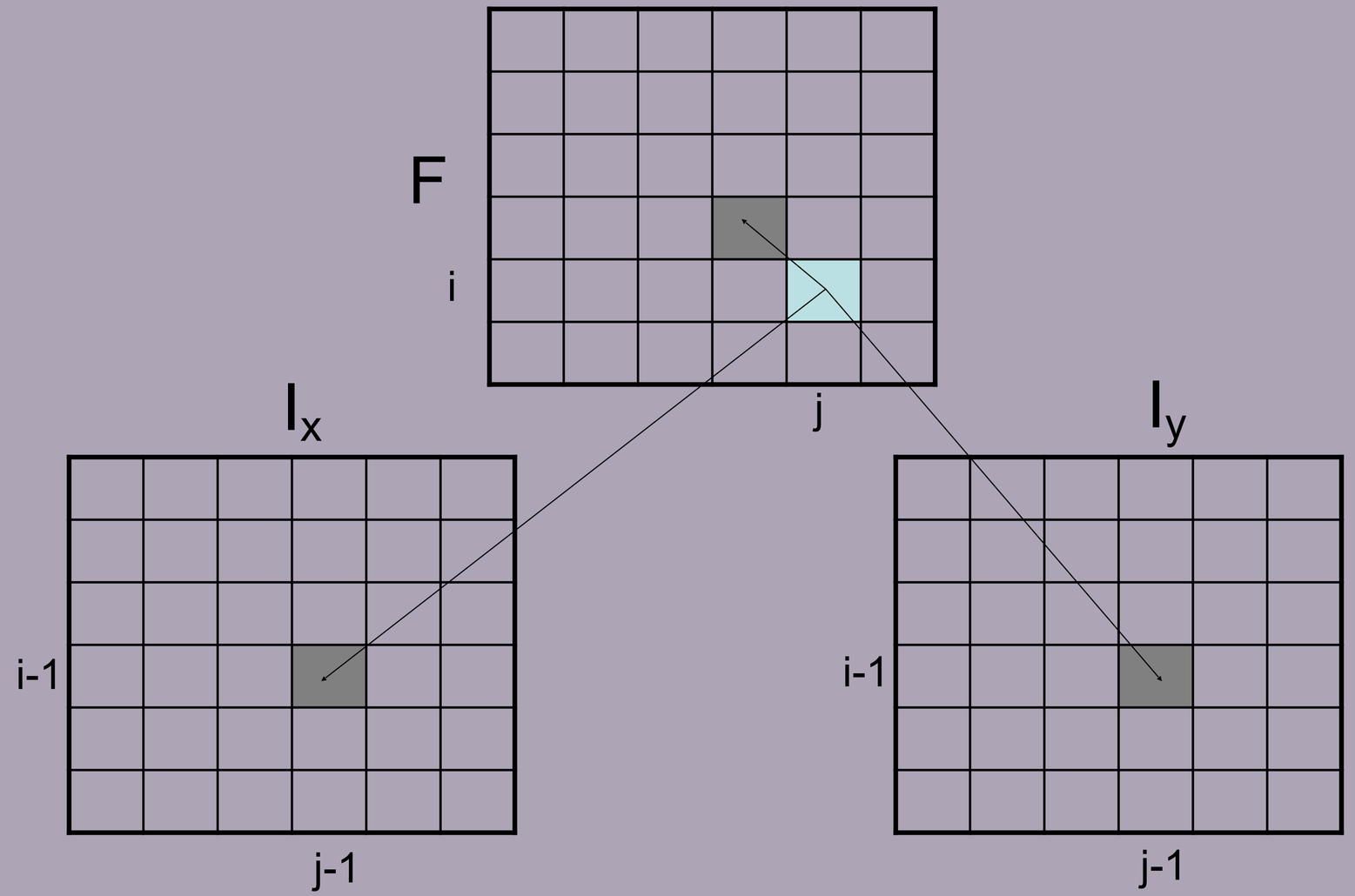


$$F(i, j) = \sigma(x_i, y_j) + \max \begin{cases} F(i-1, j-1) & \text{Continuing alignment} \\ I_x(i-1, j-1) & \text{Closing gaps in x} \\ I_y(i-1, j-1) & \text{Closing gaps in y} \end{cases}$$

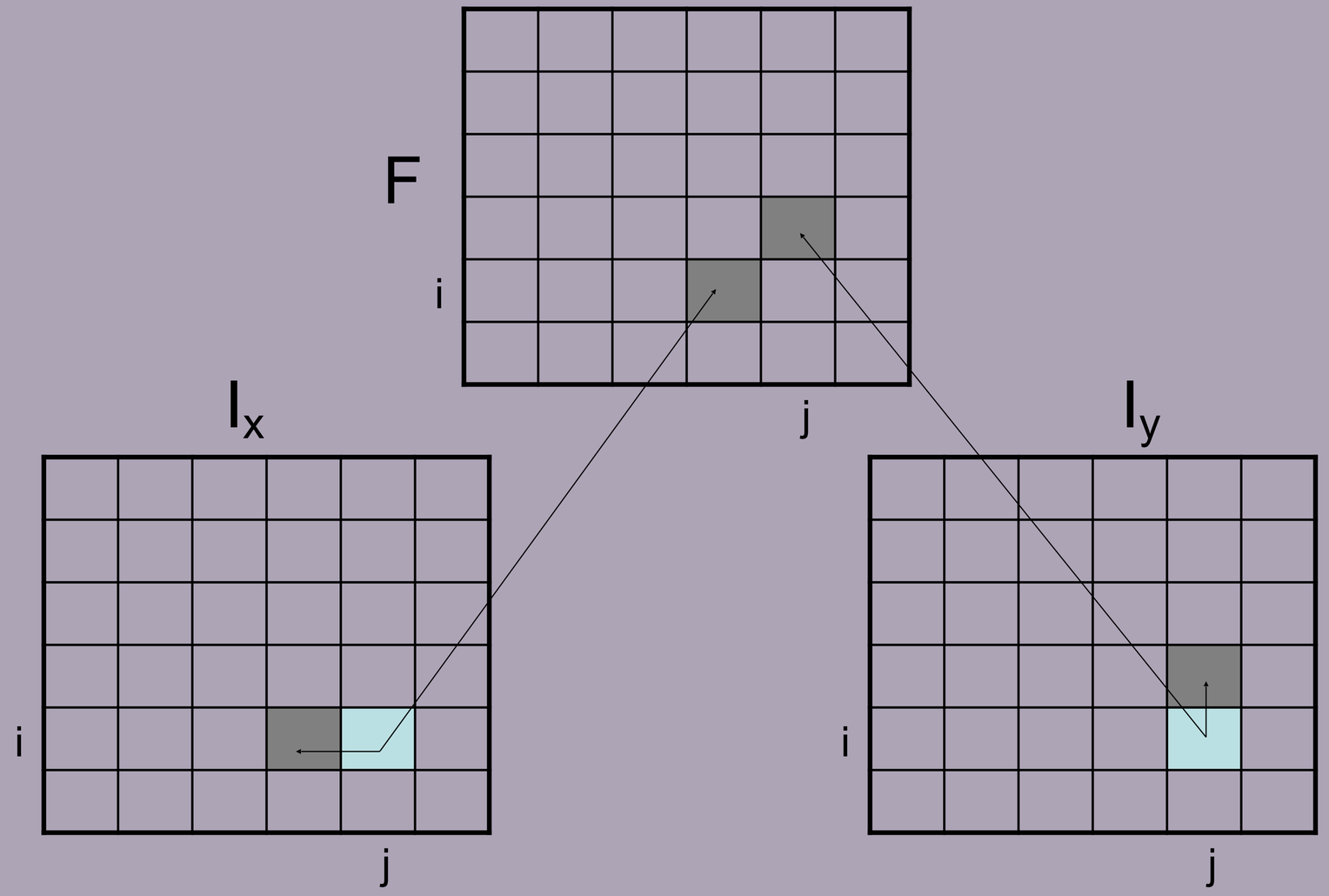
$$I_x(i, j) = \max \begin{cases} F(i, j-1) + d & \text{Opening a gap in x} \\ I_x(i, j-1) + e & \text{Gap extension in x} \end{cases}$$

$$I_y(i, j) = \max \begin{cases} F(i-1, j) + d & \text{Opening a gap in y} \\ I_y(i-1, j) + e & \text{Gap extension in y} \end{cases}$$

Data dependency

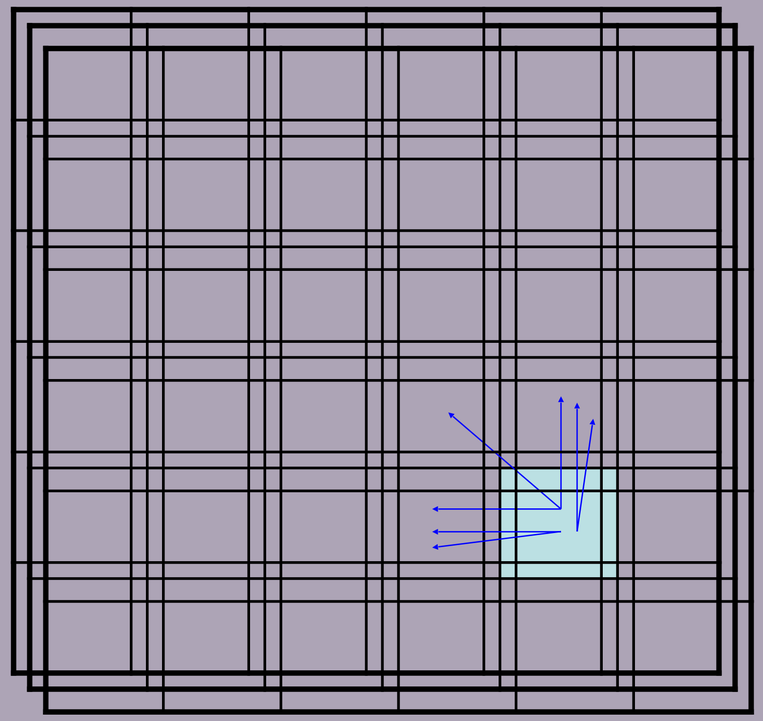


Data dependency



Data dependency

- If we stack all three matrices
  - No cyclic dependency
  - Therefore, we can fill in all three matrices in order



Algorithm

- for i = 1:m
  - for j = 1:n
    - Fill in F(i, j), I_x(i, j), I_y(i, j)
  - end

  end
- F(M, N) = max (F(M, N), I_x(M, N), I_y(M, N))

- Time: O(MN)
- Space: O(MN) or O(N) when combine with the linear-space algorithm

x = \ y =	G	C	C
	0	-∞	-∞	-∞
G	-∞			
C	-∞			
A	-∞			
C	-∞			

F: aligned on both

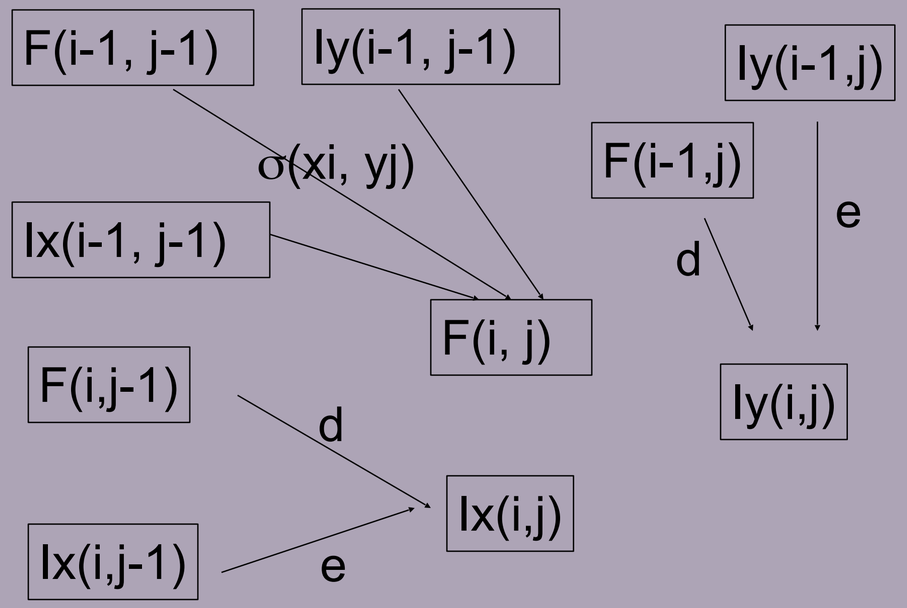
x = \ y =	G	C	C
	-∞	-∞	-∞	-∞
G	-5			
C	-6			
A	-7			
C	-8			

Iy: Insertion on y

m = 2
s = -2
d = -5
e = -1

x = \ y =	G	C	C
	-∞	-5	-6	-7
G	-∞			
C	-∞			
A	-∞			
C	-∞			

Ix: Insertion on x



x = \ y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2		
C	-∞			
A	-∞			
C	-∞			

F

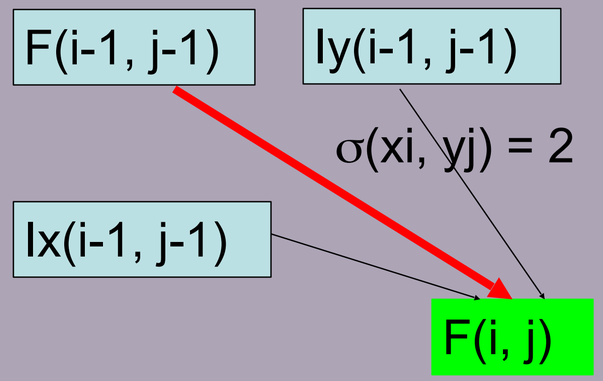
x = \ y =	G	C	C
	-∞	-∞	-∞	-∞
G	-5			
C	-6			
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

x = \ y =	G	C	C
	-∞	-5	-6	-7
G	-∞			
C	-∞			
A	-∞			
C	-∞			

Ix



	y =	G	C	C
x =	0	-∞	-∞	-∞
G	-∞	2	-7	
C	-∞			
A	-∞			
C	-∞			

F

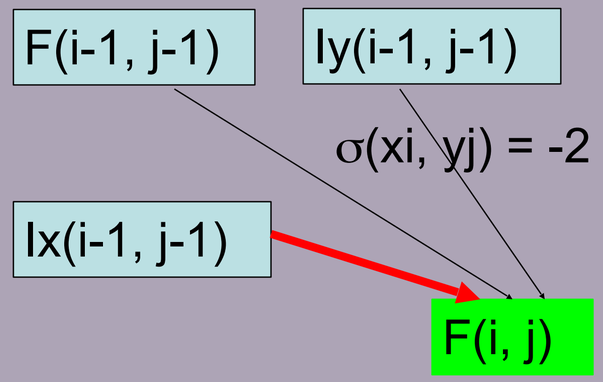
	y =	G	C	C
x =	-∞	-∞	-∞	-∞
G	-5			
C	-6			
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

	y =	G	C	C
x =	-∞	-5	-6	-7
G	-∞			
C	-∞			
A	-∞			
C	-∞			

Ix



F

x = \ y =	G	C	C
0	-∞	-∞	-∞
G	-∞	2	-7
C	-∞		
A	-∞		
C	-∞		

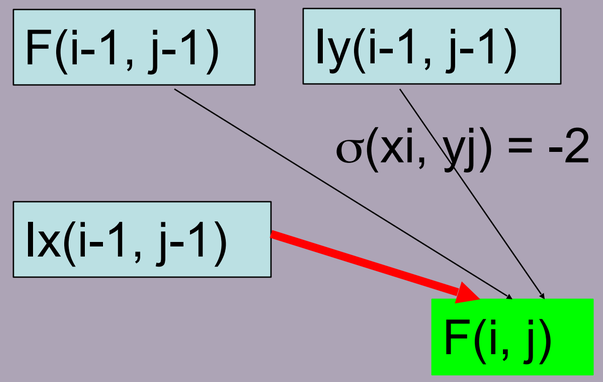
Iy

x = \ y =	G	C	C
-∞	-∞	-∞	-∞
G	-5		
C	-6		
A	-7		
C	-8		

m = 2
s = -2
d = -5
e = -1

Ix

x = \ y =	G	C	C
-∞	-5	-6	-7
G	-∞		
C	-∞		
A	-∞		
C	-∞		



y =	G	C	C
x =	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞			
A	-∞			
C	-∞			

F

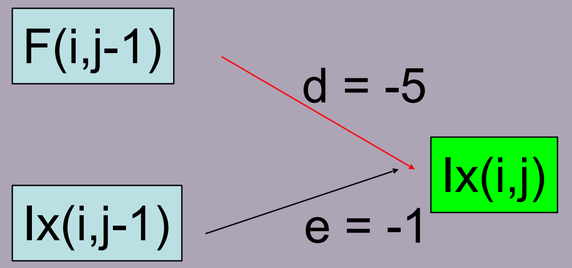
y =	G	C	C
x =		-∞	-∞	-∞
G	-5			
C	-6			
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

y =	G	C	C
x =		-5	-6	-7
G	-∞	-∞	-3	
C	-∞			
A	-∞			
C	-∞			

Ix



F

x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞			
A	-∞			
C	-∞			

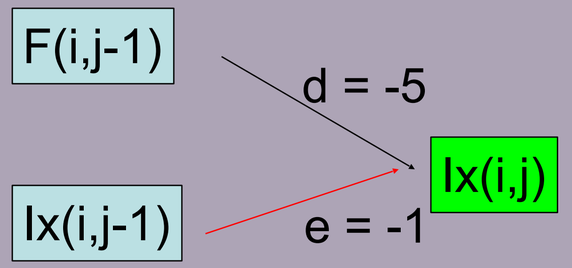
Iy

x =	y =	G	C	C
		-∞	-∞	-∞
G	-5			
C	-6			
A	-7			
C	-8			

m = 2
s = -2
d = -5
e = -1

Ix

x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞			
A	-∞			
C	-∞			



F

x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞			
A	-∞			
C	-∞			

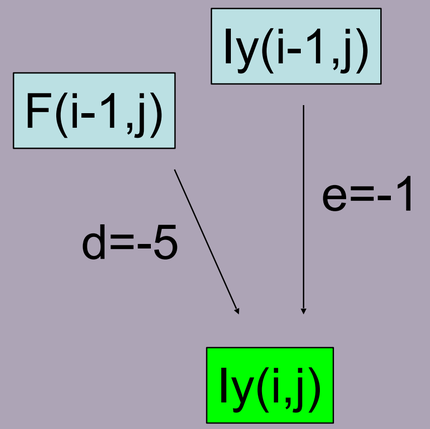
Iy

x =	y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6			
A	-7			
C	-8			

m = 2
s = -2
d = -5
e = -1

Ix

x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞			
A	-∞			
C	-∞			



x = \ y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7		
A	-∞			
C	-∞			

F

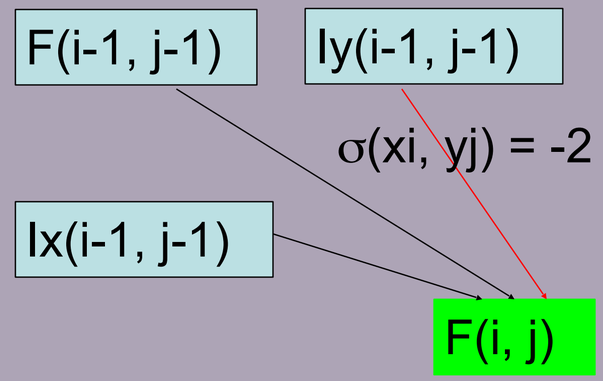
x = \ y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6			
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

x = \ y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞			
A	-∞			
C	-∞			

Ix



	y =	G	C	C
x =	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	
A	-∞			
C	-∞			

F

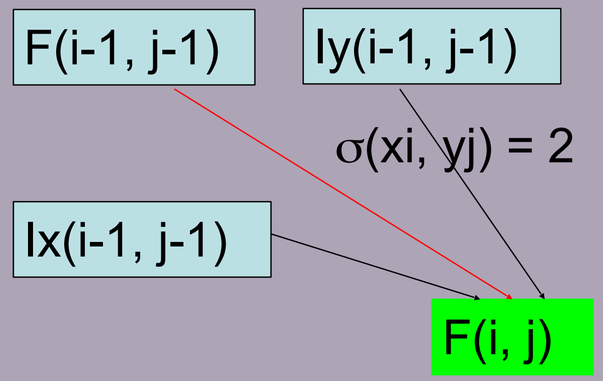
	y =	G	C	C
x =		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6			
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

	y =	G	C	C
x =		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞			
A	-∞			
C	-∞			

Ix



x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞			
C	-∞			

F

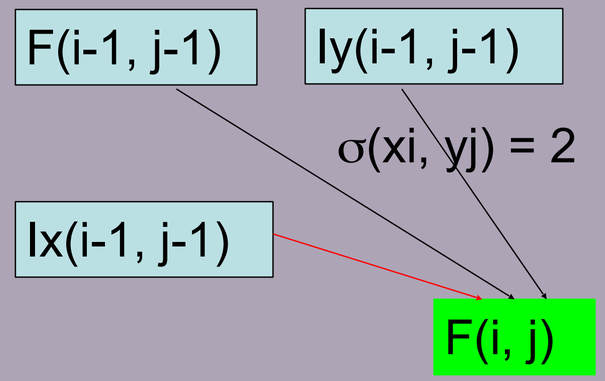
x =	y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6			
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞			
A	-∞			
C	-∞			

Ix



F

x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞			
C	-∞			

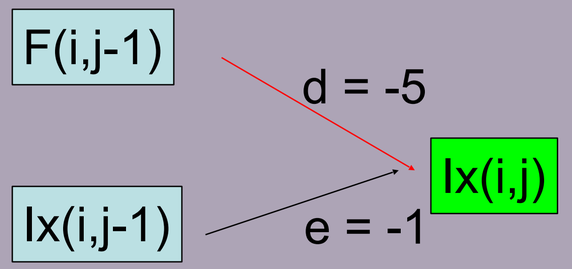
Iy

x =	y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6			
A	-7			
C	-8			

m = 2
s = -2
d = -5
e = -1

Ix

x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞			
C	-∞			



x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞			
C	-∞			

F

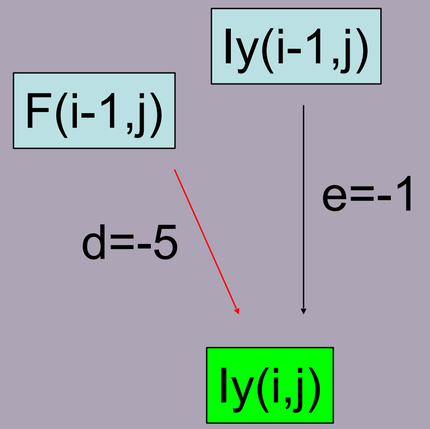
x =	y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3		
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞			
C	-∞			

Ix



F

x = \ y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞			
C	-∞			

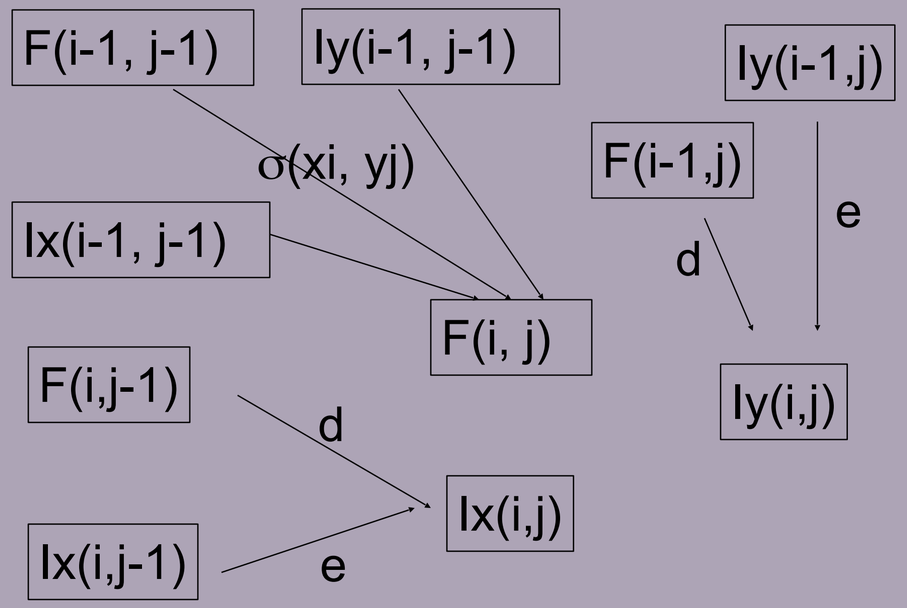
Iy

x = \ y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7			
C	-8			

m = 2
s = -2
d = -5
e = -1

Ix

x = \ y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞			
C	-∞			



x = \ y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-5
A	-∞	-8	-5	2
C	-∞			

F

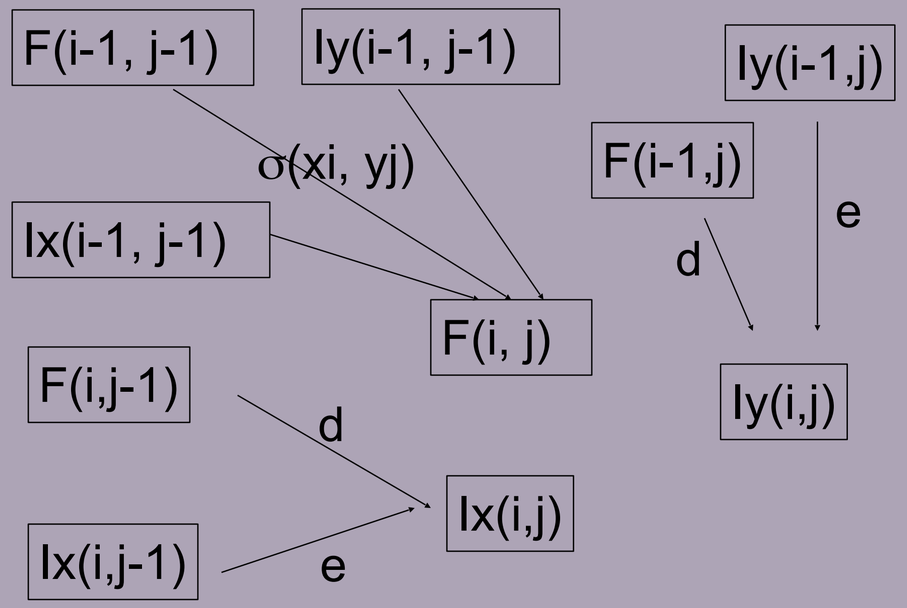
x = \ y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7			
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

x = \ y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞	-∞	-13	-10
C	-∞			

Ix



x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞	-8	-5	2
C	-∞			

F

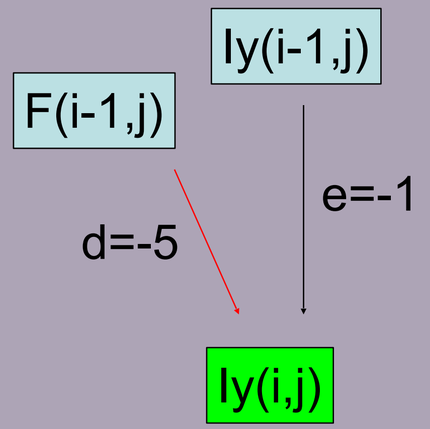
x =	y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7	-8	-1	
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞	-∞	-13	-10
C	-∞			

Ix



	y =	G	C	C
x =	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞	-8	-5	2
C	-∞			

F

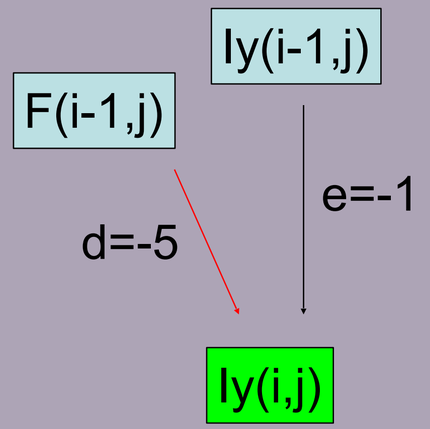
	y =	G	C	C
x =		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7	-8	-1	-6
C	-8			

Iy

m = 2
s = -2
d = -5
e = -1

	y =	G	C	C
x =		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞	-∞	-13	-10
C	-∞			

Ix



x = \ y =		G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞	-8	-5	2
C	-∞	-9	-6	1

F

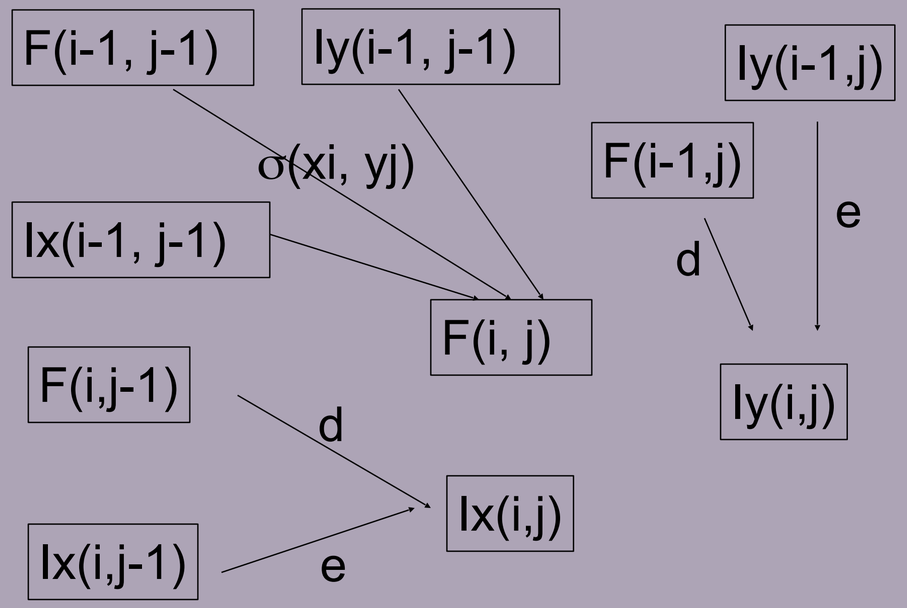
x = \ y =		G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7	-8	-1	-6
C	-8	-13	-2	-3

Iy

m = 2
s = -2
d = -5
e = -1

x = \ y =		G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞	-∞	-13	-10
C	-∞	-∞	-14	-11

Ix



x =	y = G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞	-8	-5	2
C	-∞	-9	-6	1

F

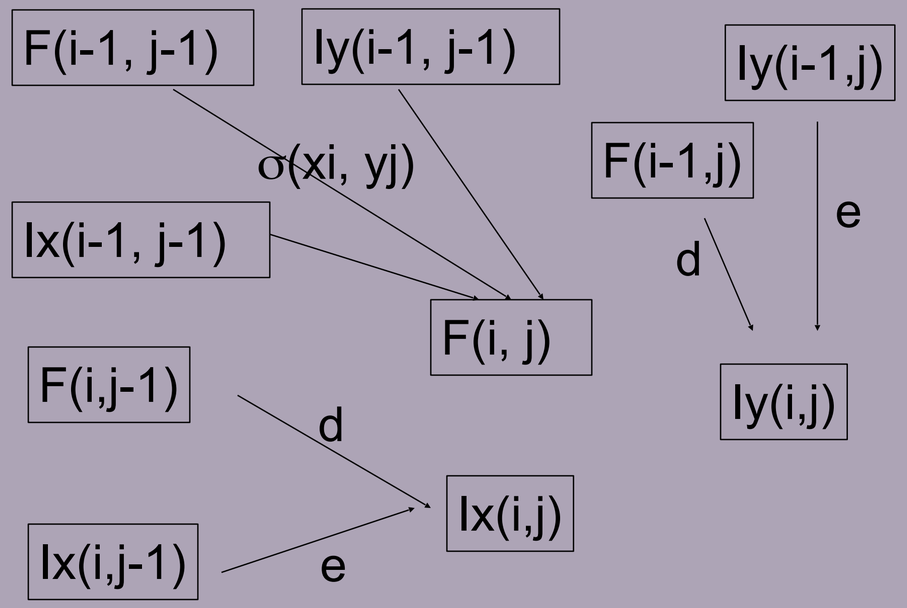
x =	y = G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7	-8	-1	-6
C	-8	-13	-2	-3

Iy

m = 2
s = -2
d = -5
e = -1

x =	y = G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞	-∞	-13	-10
C	-∞	-∞	-14	-11

Ix



x =	y =	G	C	C
	0	-∞	-∞	-∞
G	-∞	2	-7	-8
C	-∞	-7	4	-1
A	-∞	-8	-5	2
C	-∞	-9	-6	1

F

x GCAC

| |

y GC−C

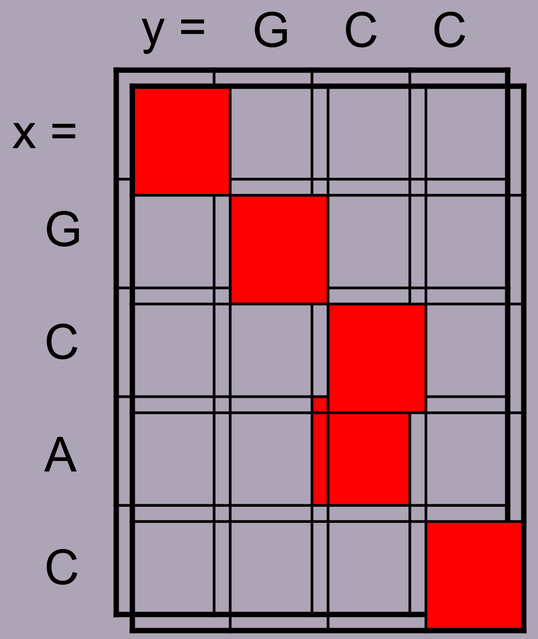
x =	y =	G	C	C
		-∞	-∞	-∞
G	-5	-∞	-∞	-∞
C	-6	-3	-12	-13
A	-7	-8	-1	-6
C	-8	-13	-2	-3

Iy

m = 2
s = -2
d = -5
e = -1

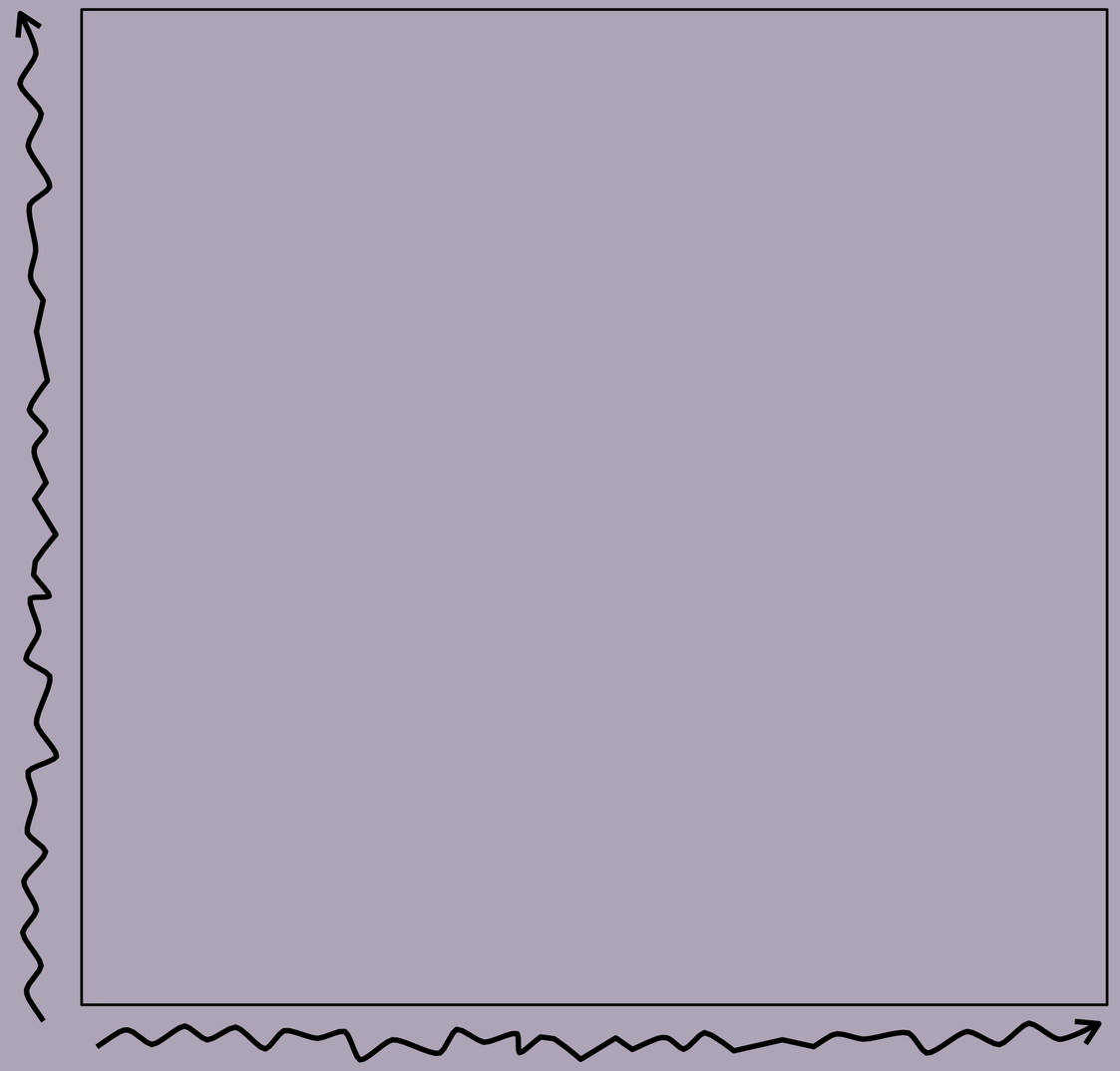
x =	y =	G	C	C
		-5	-6	-7
G	-∞	-∞	-3	-4
C	-∞	-∞	-12	-1
A	-∞	-∞	-13	-10
C	-∞	-∞	-14	-11

Ix

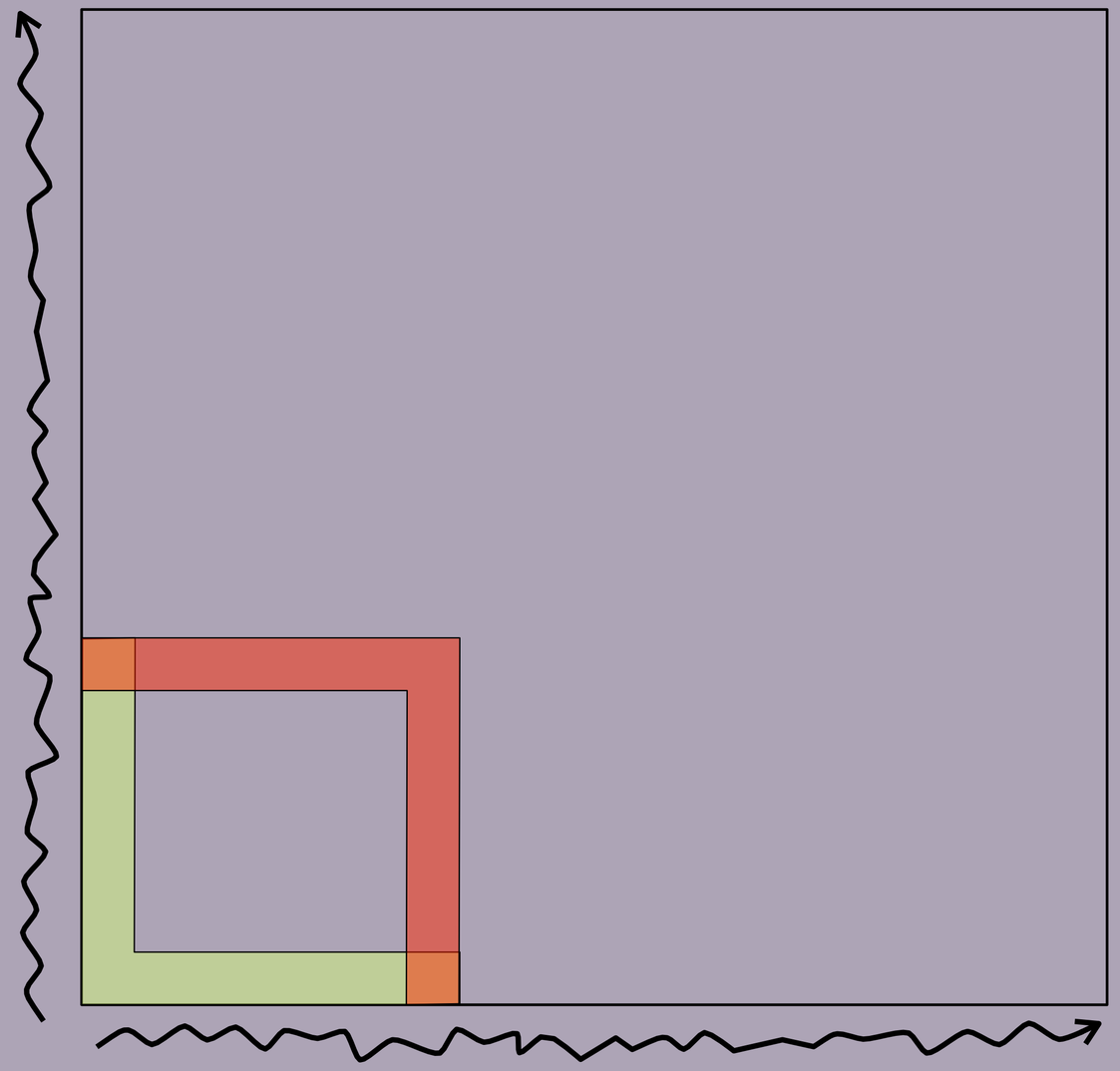


Four Russians speedup for edit distance

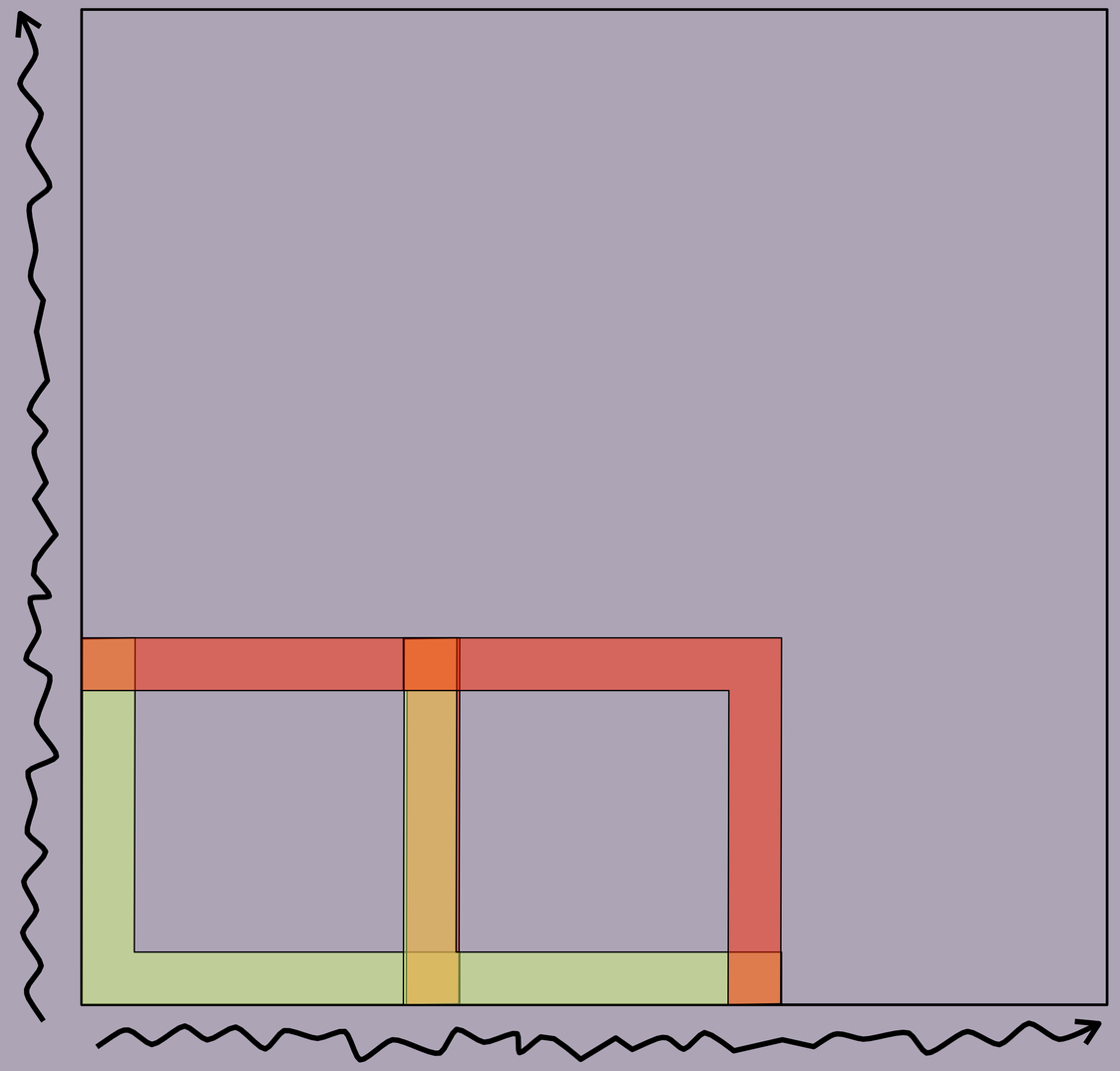
Block Edit Distance



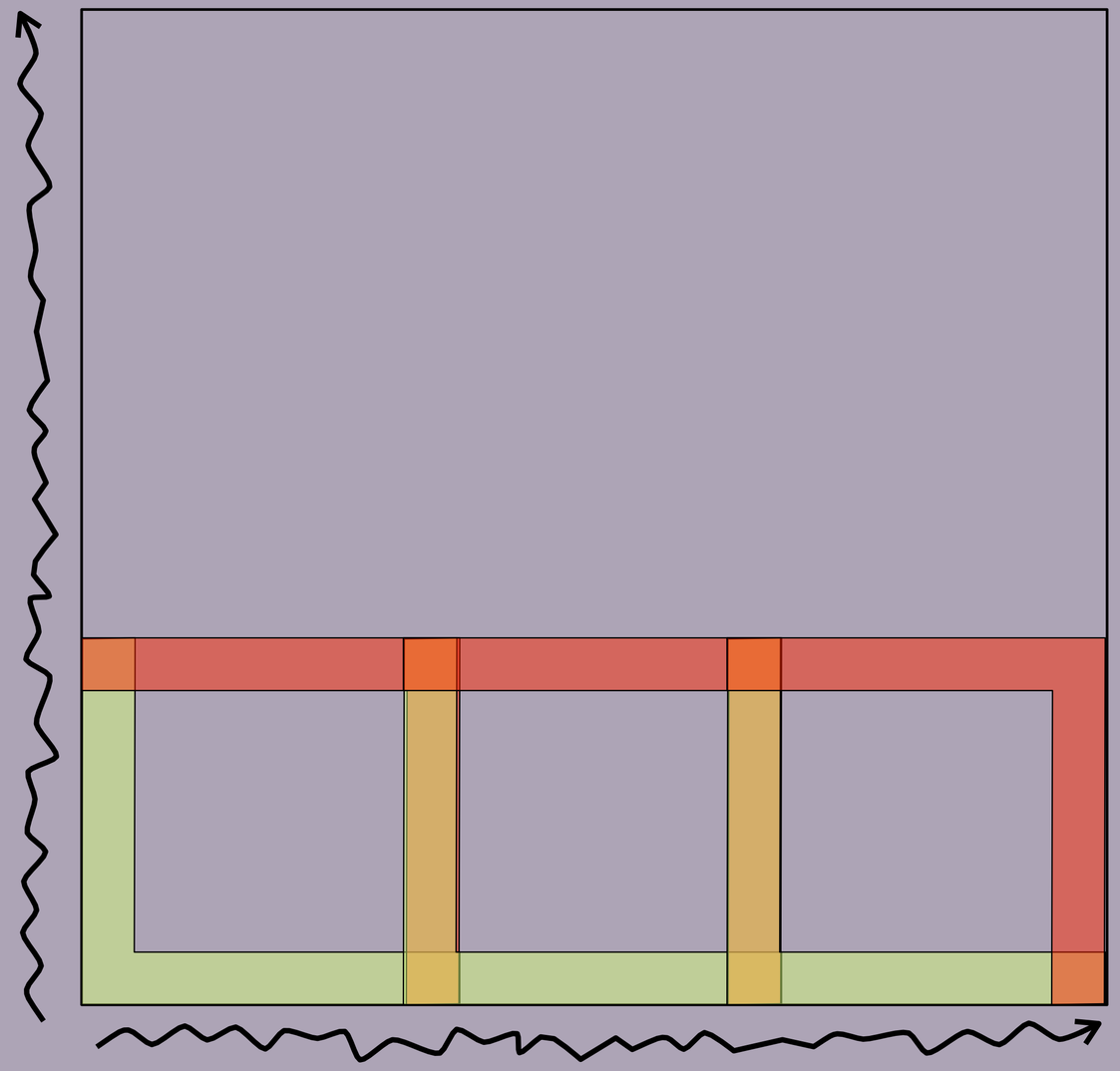
Block Edit Distance



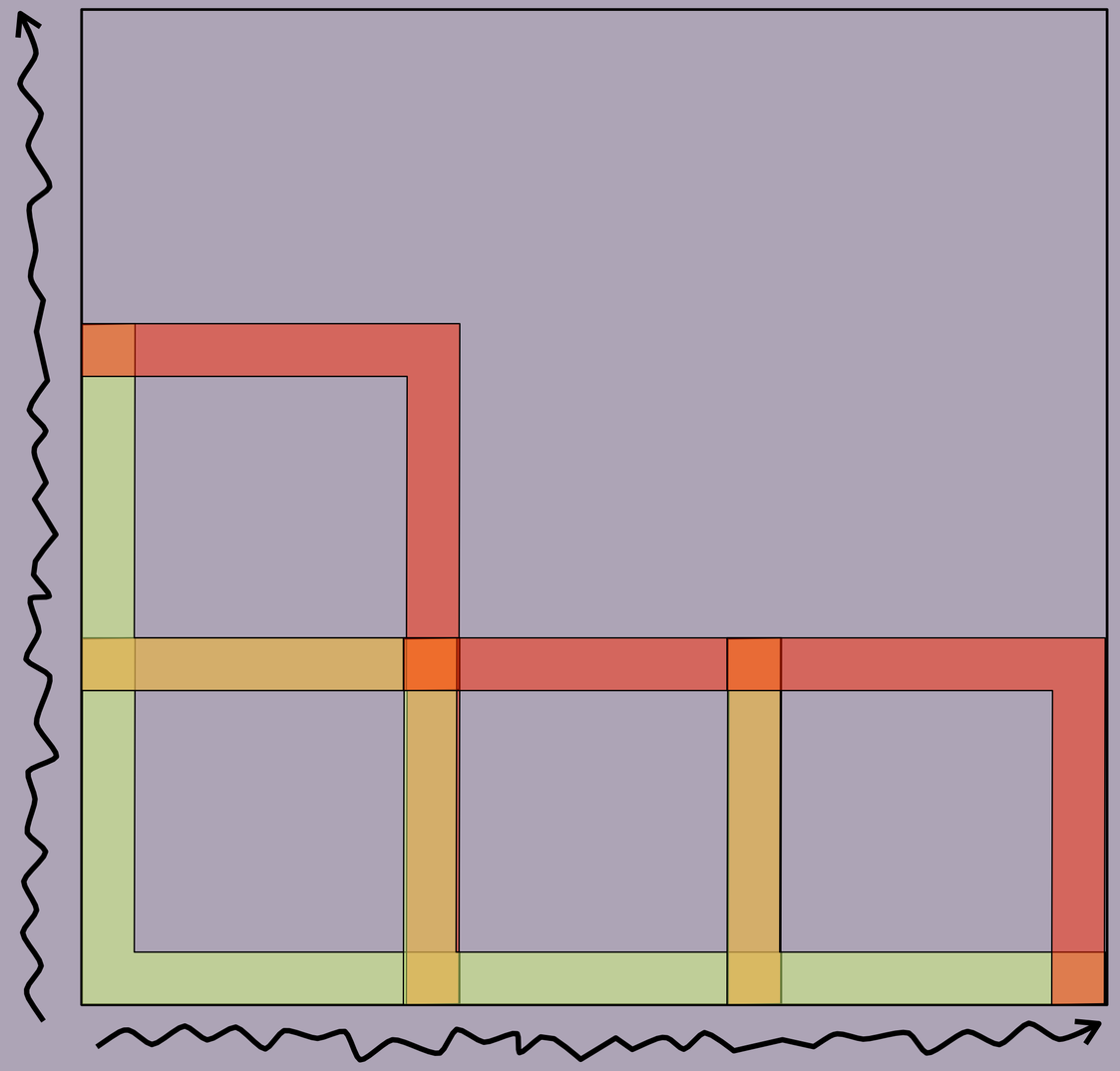
Block Edit Distance



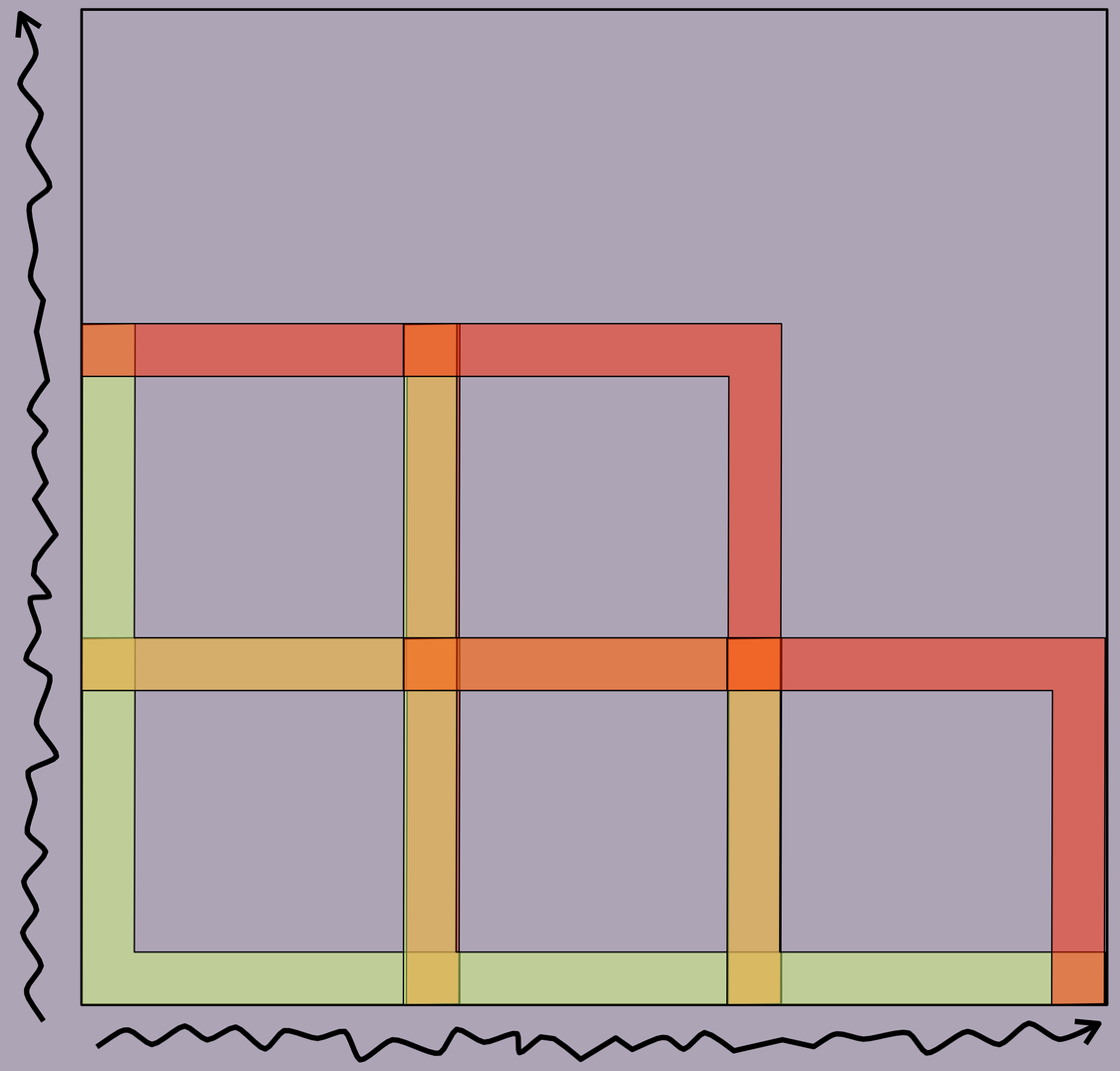
Block Edit Distance



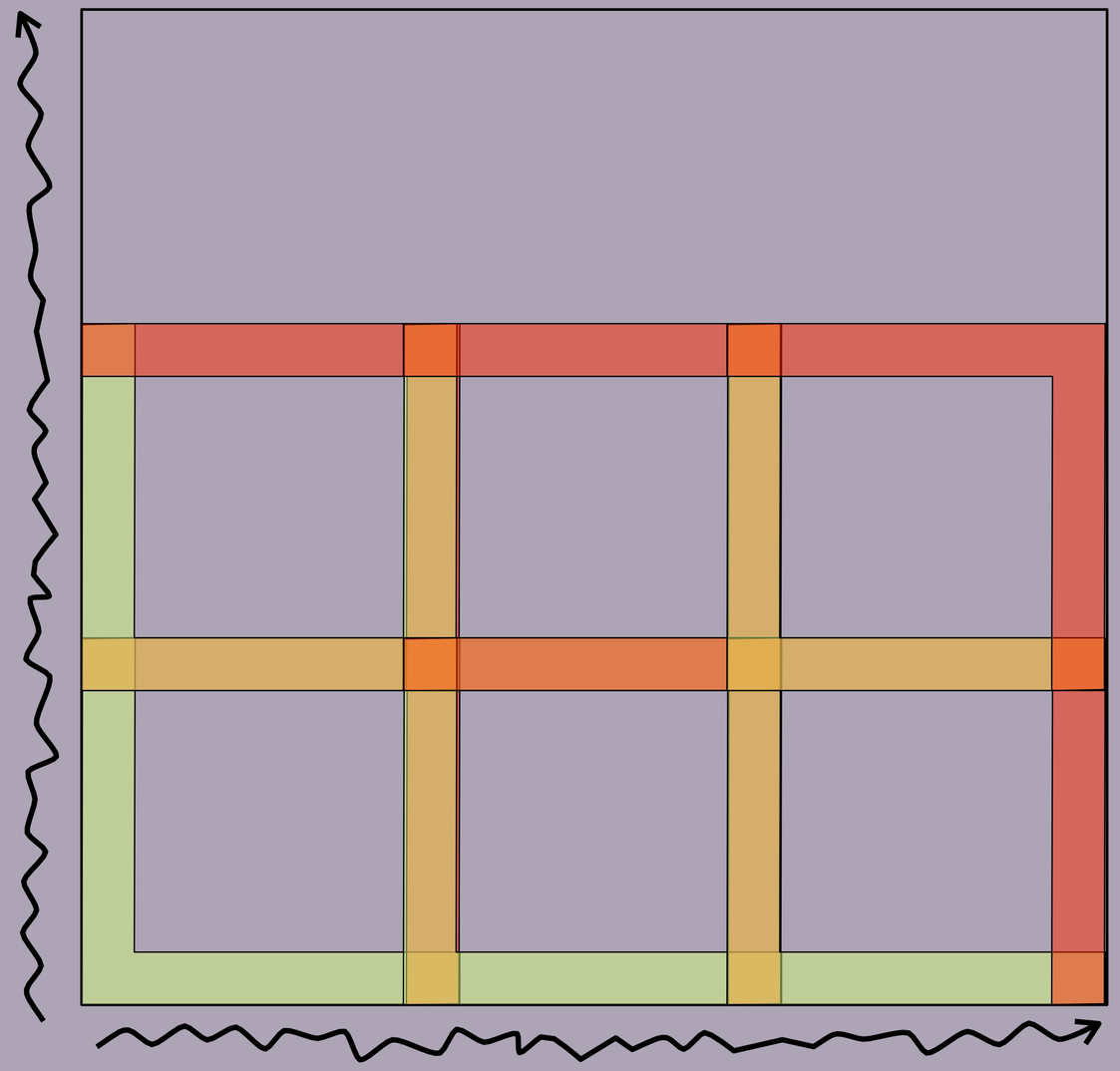
Block Edit Distance



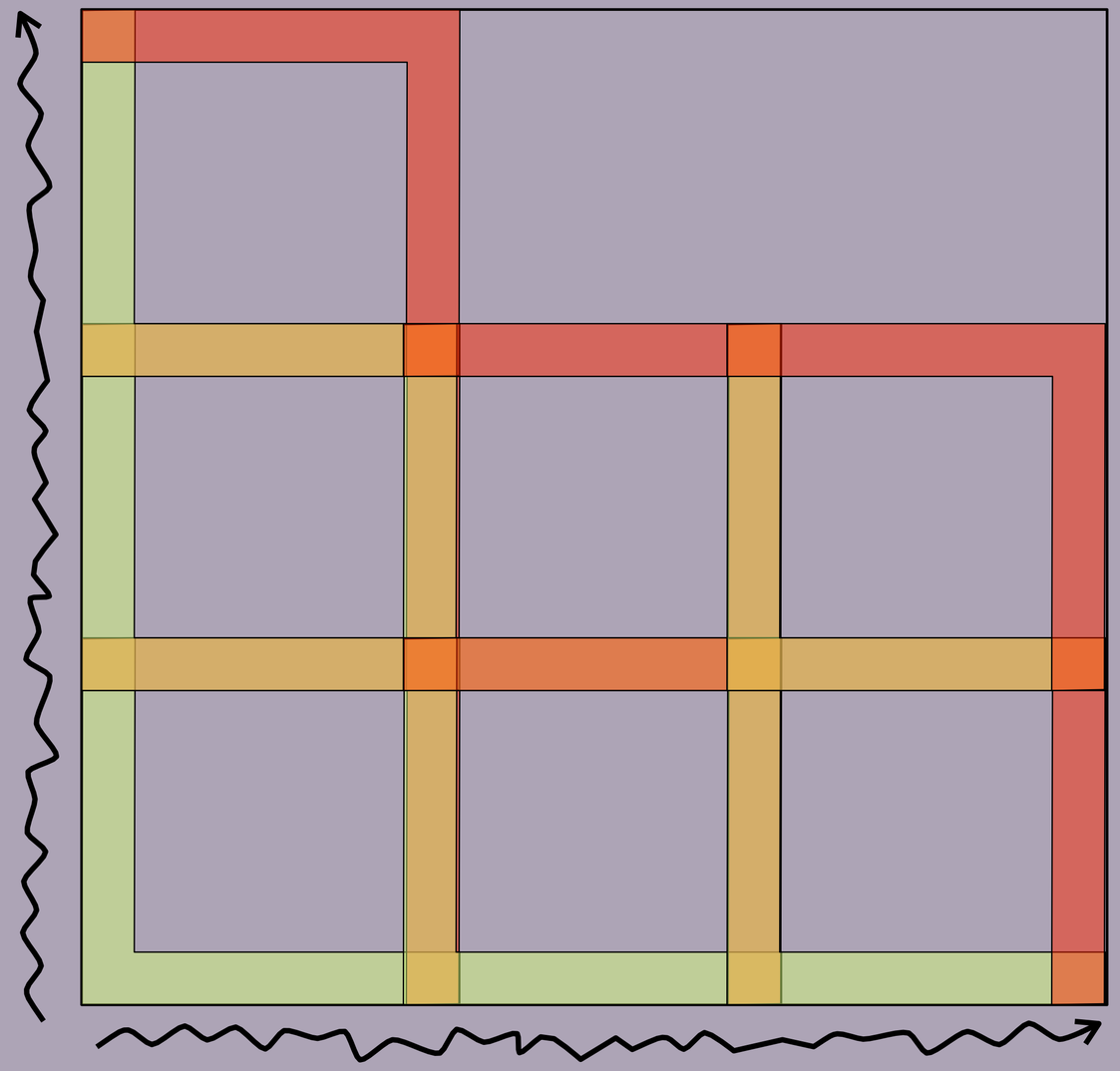
Block Edit Distance



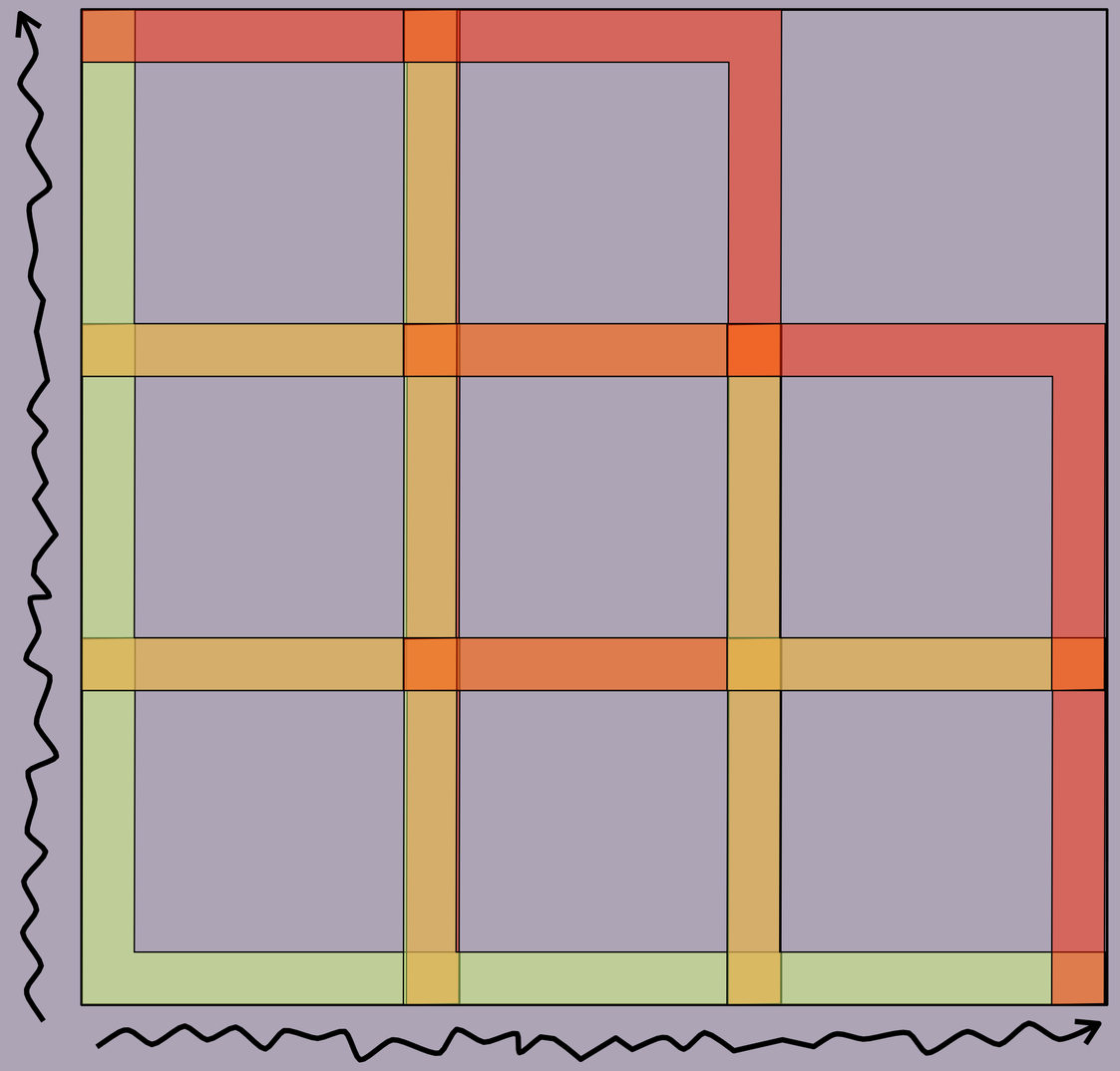
Block Edit Distance



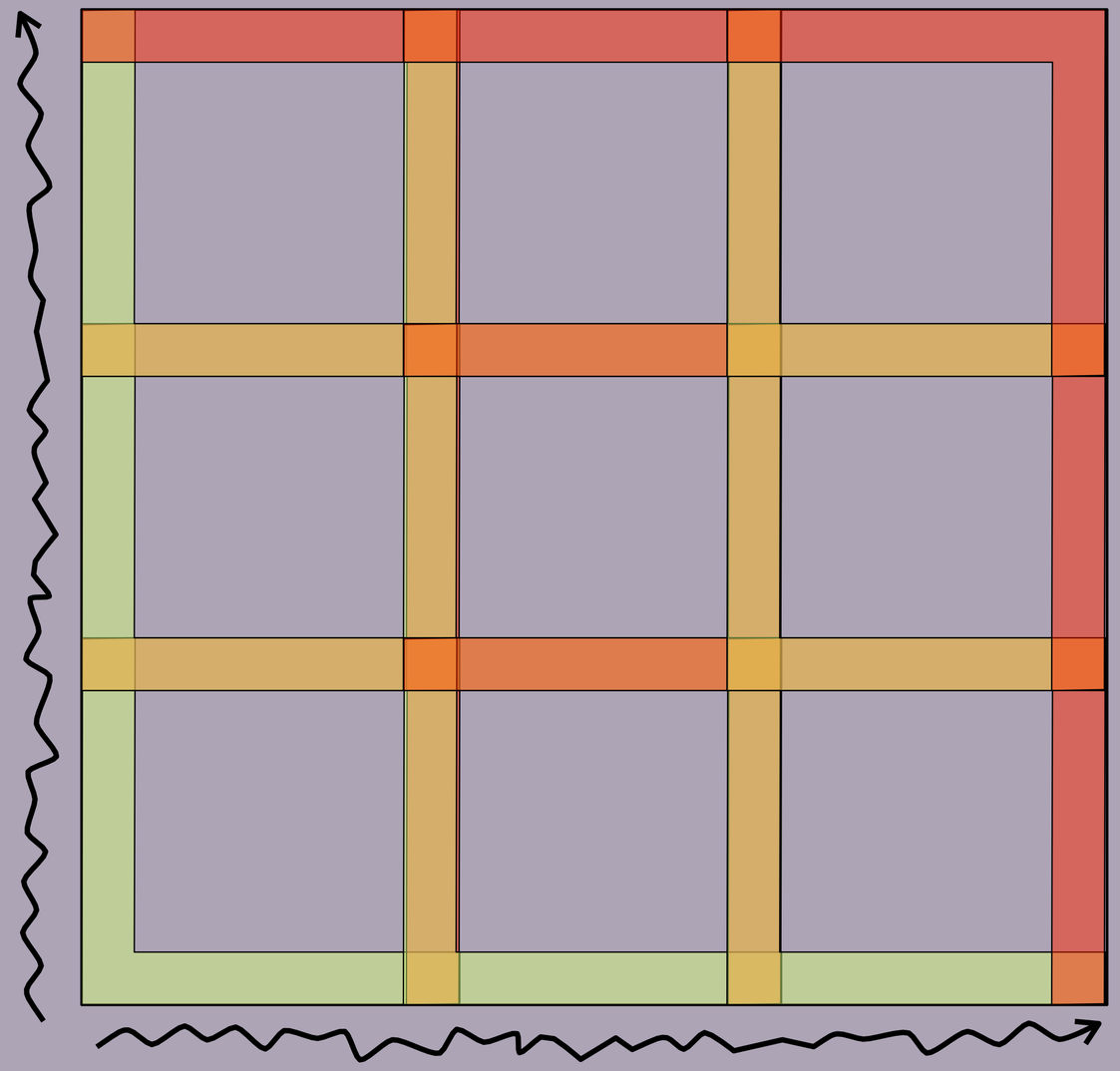
Block Edit Distance



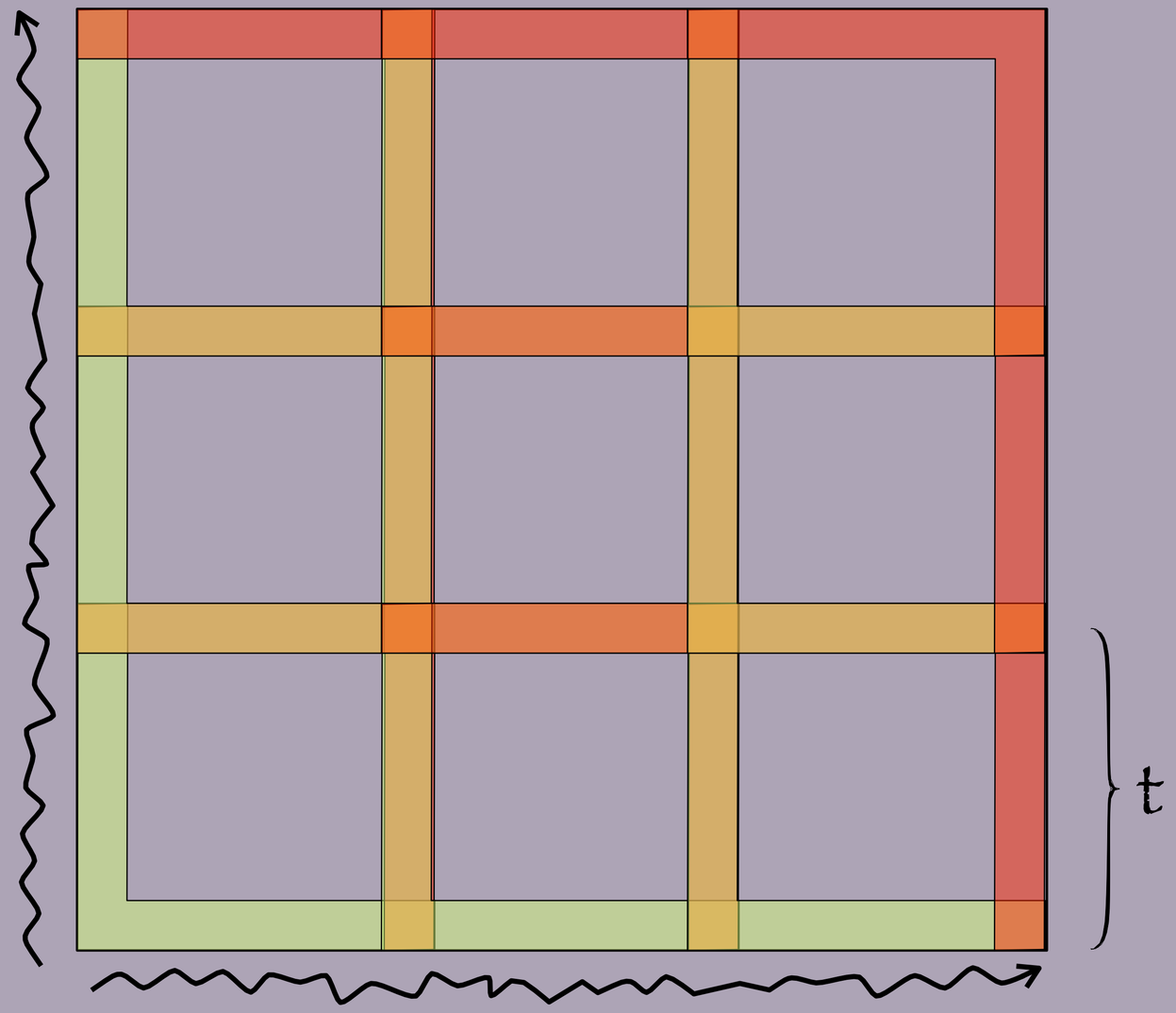
Block Edit Distance



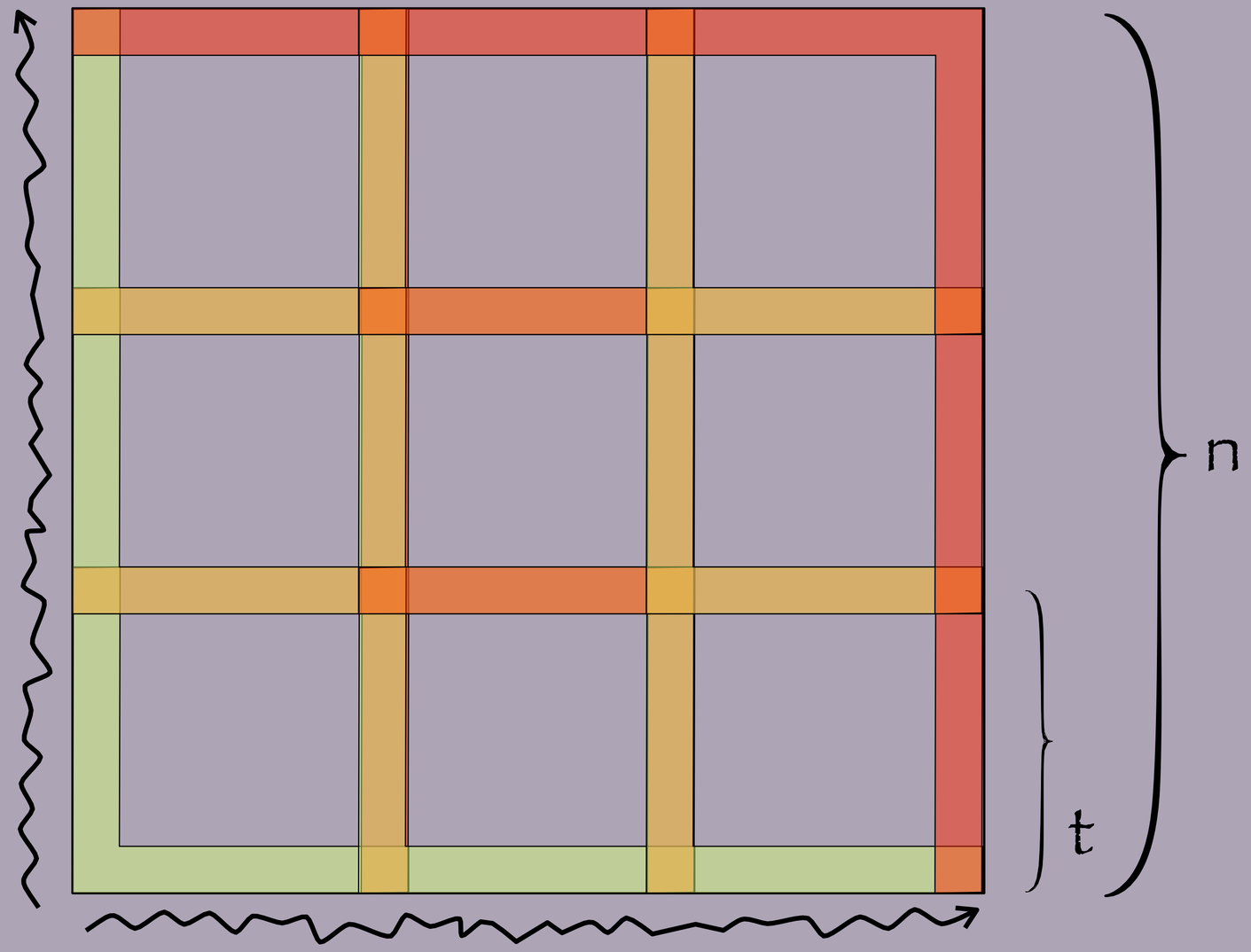
Block Edit Distance



Block Edit Distance

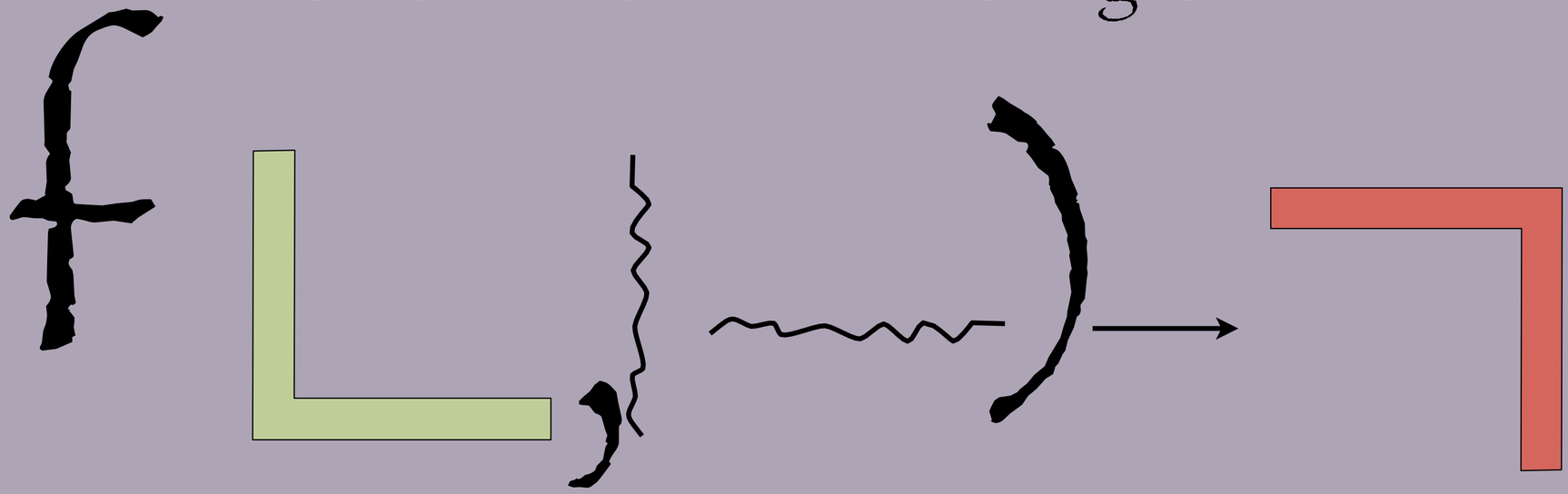


Block Edit Distance



Block Function

Assume we have a function of the following form:



If we can compute f faster than $O(t^2)$, we win.

We will see how to compute it in $O(t)$ time.

Assumptions

We're computing the plain edit distance: gaps and mismatches cost 1 and matches cost 0.

The alphabet Σ is a constant size.

$n = k(t-1)$ for some k (that is the blocks perfectly tile the matrix, with a single overlapping row and column between each adjacent pair)

Precomputing f

The way we compute f fast is to precompute f(x) for all possible x = (, ,).

How may different x values are there?

$$\underbrace{(n+1)^{2t}}_{\text{Every cell contains a number between 0 and n.}}\underbrace{|\Sigma|^{2t}}_{\text{This many pairs of strings, each of length t.}}$$

Computing each would take $O(t^2)$ time, taking in total $O((n+1)^{2t}|\Sigma|^{2t}t^2) = O(n^2)$ time. Bad!

The trick to making it work is realizing that in fact there are fewer possible functionally different inputs to x.

Offset Encoding

The elements of the rows and columns in the input are not independent.

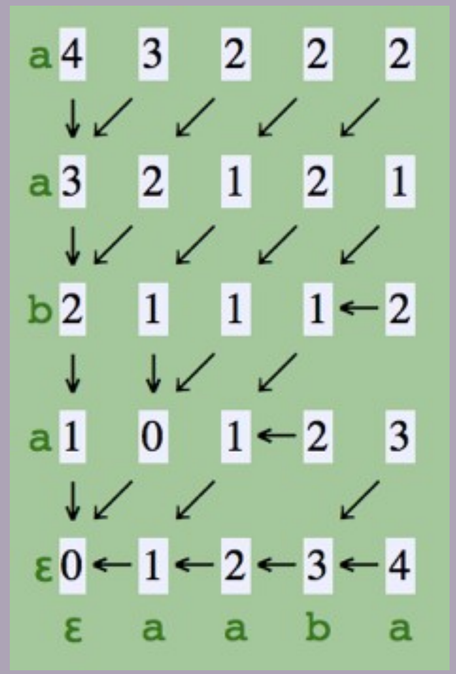
Notation. D is the matrix and D(i,j) is the value at position i,j.

Lemma. Adjacent values of D in a row, column, or diagonal differ by at most 1.

Consider element q of row i:

- $D(i,q) \leq D(i,q-1)+1$ because we can always insert a gap if we wanted to.
- Suppose we throw away character q to consider D(i, q-1):
  - If character q is matched, the edit distance increases by ≤ 1 (we can align what is was matched to against a gap):

    $D(i,q-1) \leq D(i,q)+1$
  - If character q is not matched, the edit distance goes down (by 1 since we eliminate a gap): $D(i,q-1) \leq D(i,q)$
  - Therefore: $D(i,q-1) - 1 \leq D(i,q)$



Offset Encoding, II

Can encode a row of the matrix as an initial value plus a sequence of -1,0,1:

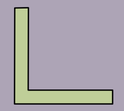
Example. *567767* → 5110 -11

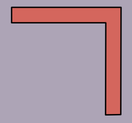
Definition. An offset vector is the encoding of a row or column as above, except that the first entry is set to 0.

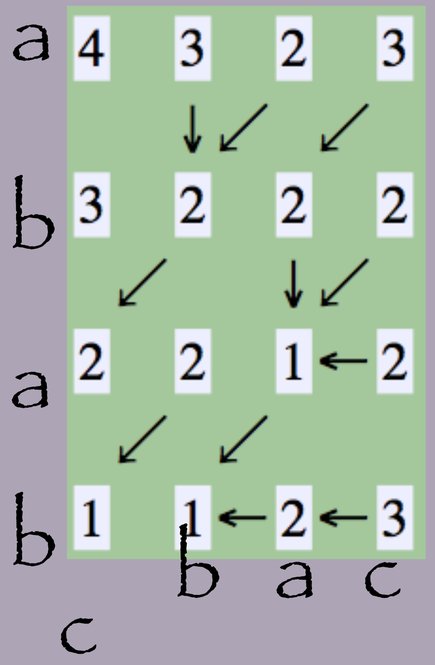
Example. *567767* → 01100 -11

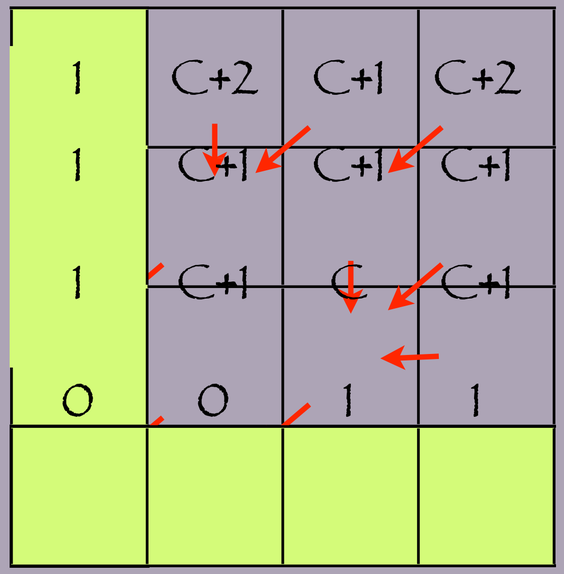
So: given the first value C and the offset vector, you can reconstruct the row or column.

Offset Encoding III

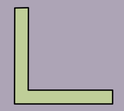
Thm. Given only the offset vectors of  and 

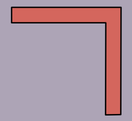
one can compute the offset vectors of 

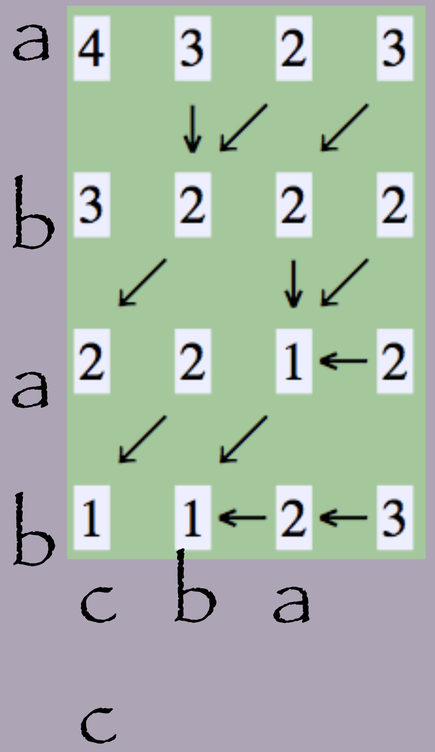


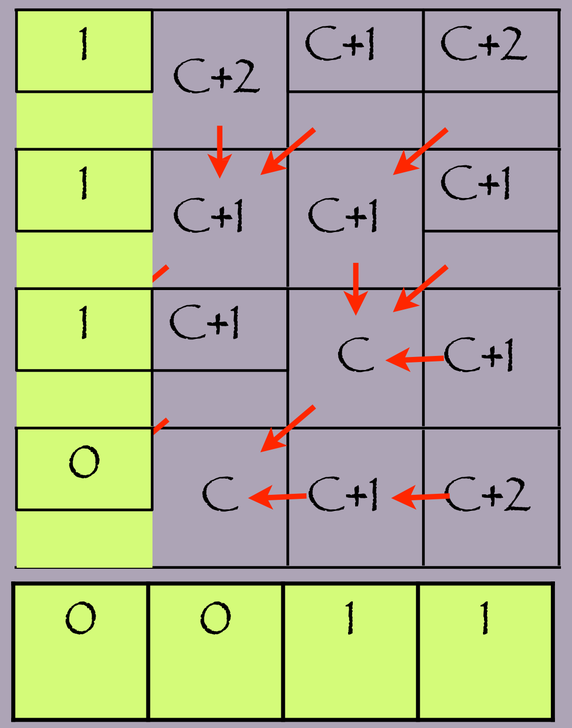


Offset Encoding III

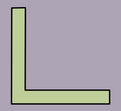
Thm. Given only the offset vectors of  and 

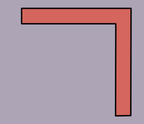
one can compute the offset vectors of 



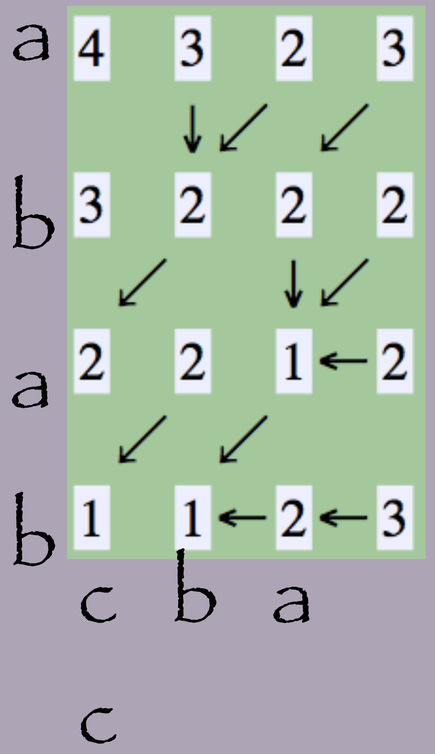


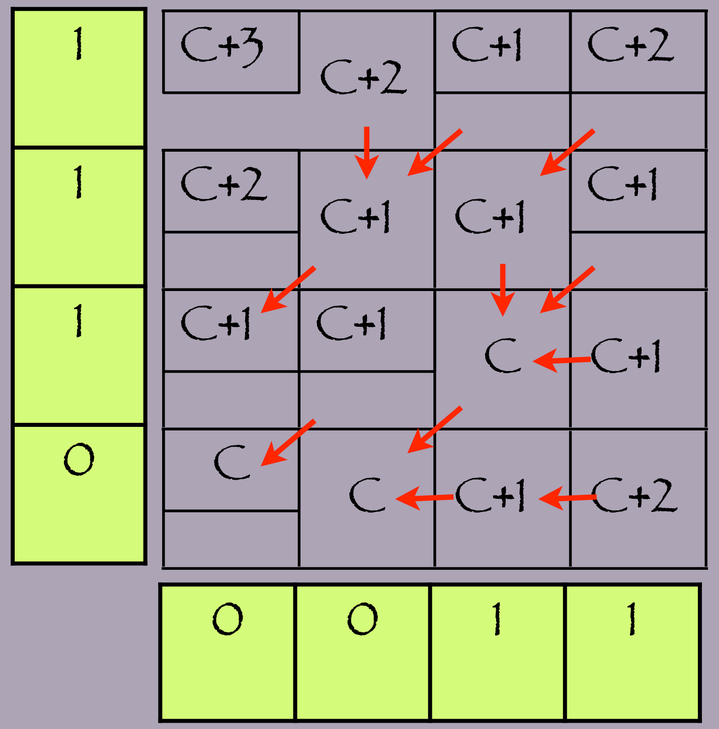
Offset Encoding III

Thm. Given only the offset vectors of  and 

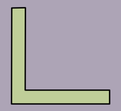


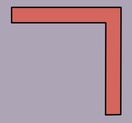
one can compute the offset vectors of

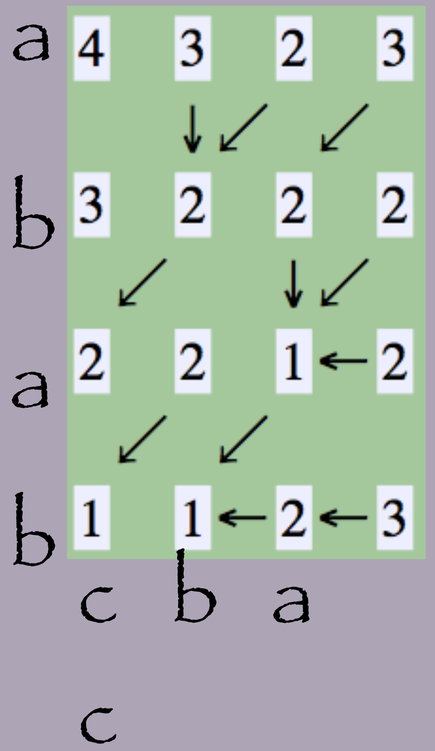


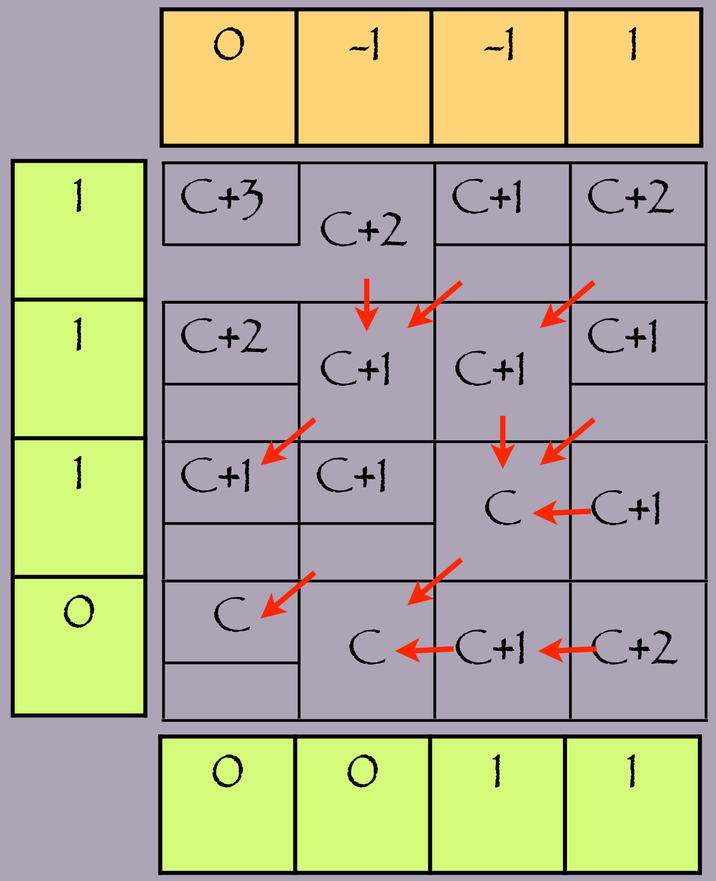


Offset Encoding III

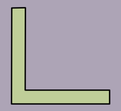
Thm. Given only the offset vectors of  and 

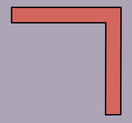
one can compute the offset vectors of 

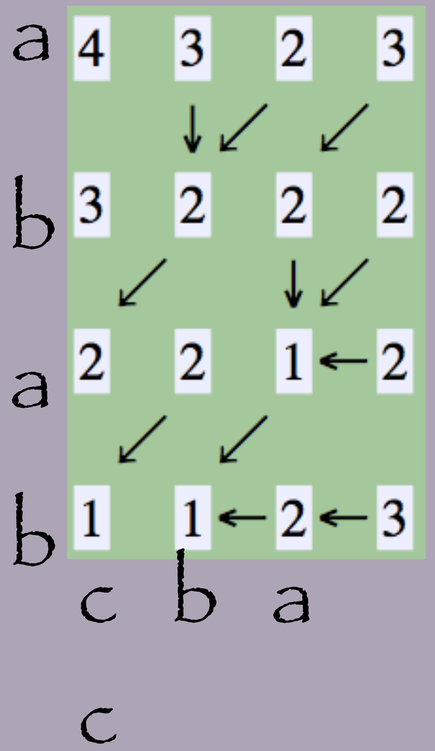


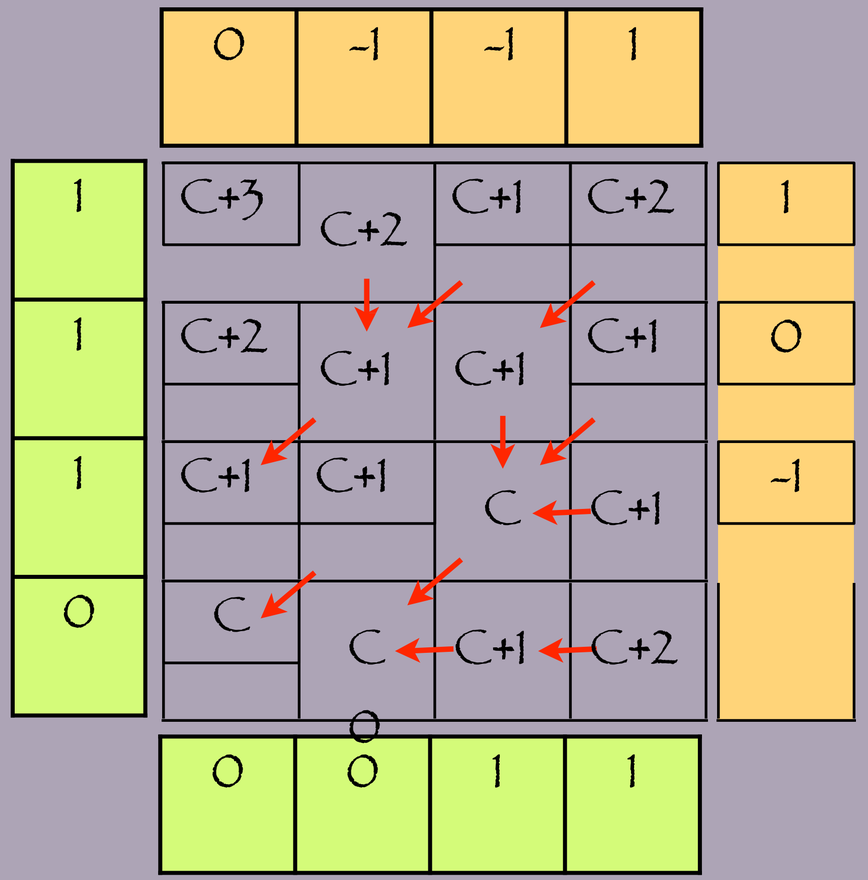


Offset Encoding III

Thm. Given only the offset vectors of  and 

one can compute the offset vectors of 





Preprocessing Time

There are $2^{2(t-1)}$ offset vectors.

There are $2^{2(t-1)}|\Sigma|^{2t}$ possible inputs x to f.

Computing all values of f(x) takes now time $O((2|\Sigma|)^{2t} t^2)$.

Setting $t = \log_{2|\Sigma|} n$, this becomes $O(n(\log n)^2)$

Storing f for quick access

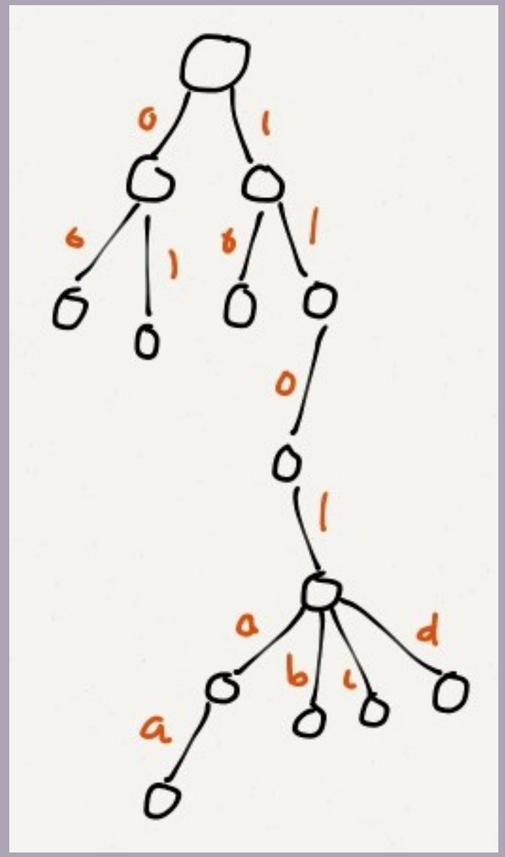
We have $2^{2(t-1)}|\Sigma|^{2t}$ possible inputs x to f.

How do we store the values f(x) so we can access f(x) in time O(t)?

Storing f for quick access

We have $2^{2(t-1)}|\Sigma|^{2t}$ possible inputs x to f.

How do we store the values f(x) so we can access f(x) in time O(t)?



Depth ≈ 3t ≈ O(t)

Total Runningtime

We have $O(n^2 / t^2)$ blocks to compute.

Accessing f(x) for each takes time $O(t)$, so our time to "fill in" the matrix is $O(tn^2/t^2) = O(n^2/t)$

With $t = O(\log n)$ the total timeis:

$$O(n^2 / \log n + n(\log n)^2) = O(n^2 / \log n)$$ FTW!

(In the RAM model, where we can access things of size log n in constant time, we get the even better time of $O(n^2 / \log^2 n)$)

In Practice

Often useful to take t = some constant instead of log n.

Doesn't give you an asymptotic speed up, but now runs in time $O(n^2 / t)$ so the constant factor is better.